O'REILLY®

TURING 图灵程序设计丛书



深入理解SVG

SVG Colors, Patterns & Gradients

[美] Amelia Bellamy-Royds, Kurt Cagle 著

刘涛 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

刘涛

网络常用名“武官尚书”，前端开发工程师，目前就职于奇虎360搜索团队，曾在多个平台翻译、原创前端相关文章。热爱前端，热爱翻译，关注前端技术的发展变迁，热衷于新技术的学习研究。

图灵程序设计丛书

深入理解SVG

SVG Colors, Patterns & Gradients

[美] Amelia Bellamy-Royds Kurt Cagle 著
刘涛 译

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北京

图书在版编目（CIP）数据

深入理解SVG / (加) 艾美利亚·拜拉梅-罗兹
(Amelia Bellamy-Royds) ，(美) 科特·卡戈
(Kurt Cagle) 著 ; 刘涛译. -- 北京 : 人民邮电出版社,
2017.10
（图灵程序设计丛书）
ISBN 978-7-115-46758-4

Ⅰ. ①深… Ⅱ. ①艾… ②科… ③刘… Ⅲ. ①图形软
件 Ⅳ. ①TP391.412

中国版本图书馆CIP数据核字(2017)第213786号

内容提要

本书介绍 SVG 绘画，包括基础知识和如何通过混合和搭配工具来生成复杂的效果。主要内容有：把 SVG 代码转换为可视图形的渲染模型，如何使用颜色，透明度的控制方法以及它对图片的影响，渲染服务和渐变。

本书适合所有想利用 SVG 提高 Web 体验的读者。

◆ 著　[美] Amelia Bellamy-Royds Kurt Cagle
译　刘　涛
责任编辑　朱　巍
执行编辑　付　阳
责任编辑　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：880×1230　1/32
印张：7.875
字数：283千字　2017年10月第1版
印数：1－3 000册　2017年10月北京第1次印刷

著作权合同登记号　图字：01-2017-4860号

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会聚集了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

目录

前言

本书深入讨论了 SVG 的一个特定的方面：绘画。这里的绘画使用的不是油墨或水彩，而是可被计算机转化为有色像素的图形指令。本书探讨了创建它的可能性以及潜在的风险。书中不仅描述了一些基础知识，还介绍了如何通过混合和搭配工具来生成复杂的效果。

本书起源于一个介绍如何在 Web 上使用 SVG 的项目。为了保证本书篇幅适中，并适合入门读者，许多细节和复杂的部分我们不再赘述。但这些细节和复杂内容会使作为图片格式的 SVG 更加丰富、美妙。掌握了 SVG 基础后，你就可以考虑去制作更加复杂的绘画和更加细致的效果。

本书内容

如果你正在阅读本书，我们希望你已经熟悉了 SVG 的基础知识，例如如何把图形定义为一系列形状，如何将图形作为一个单独的图片文件或作为 HTML 页面中的标记来使用。如果你不确定是否已经准备好，第 1 章会带你回顾一些应该知道的基础概念。

本书的剩余部分着重讲解 SVG 颜色、图案以及渐变。

- 第 2 章讨论了把 SVG 代码转换为可视图形的渲染模型，还介绍了可以设置在形状和文本上以控制它们如何绘制在屏幕上的基础属性。
- 第 3 章着重讲解了颜色：它在自然界中是如何工作的，在计算机中又是如何工作的，以及如何在 SVG 代码中指定颜色。
- 第 4 章中讨论了透明度，更确切地说，是不透明度；还介绍了控制图形不透明度的多种方法，以及它们是如何影响最终效果的。

- 第 5 章中介绍了渲染服务的概念：定义其他 SVG 图形和文本如何被绘制到屏幕上的复杂图形内容。这一章还介绍了纯色渲染服务，你最初可能会觉得它用处不大，但实际上却很有用。
- 第 6 章着眼于渐变，主要讲解了通过调整结点的位置和属性来创建不同的颜色过渡效果。
- 第 7 章探讨了如何控制线性渐变在要渲染的形状内移动。
- 第 8 章介绍了重复线性渐变以及利用它们可以实现的一些效果，同时还介绍了一些在 HTML 页面中使用渐变（以及其他渲染服务）生成内联 SVG 图标的示例和小技巧。
- 第 9 章着眼于径向渐变，包括重复径向渐变，最后还给出了一些使用多个渐变来创建复杂效果的例子。
- 第 10 章介绍了 `<pattern>`，它用于创建重复的磁贴和纹理。
- 第 11 章展示了如何使用图案来定义用于填充形状或文本的单个图像或图形。
- 第 12 章详细介绍了渲染服务是如何应用到文本上的。
- 第 13 章讨论了使用渲染服务绘制描边区域而非填充区域时出现的一些问题。
- 第 14 章给出了一些给渲染服务添加动画的例子，还讨论了在 SVG 中可用的不同添加动画方法的优点与局限。

本书最后的两个附录提供了把理论用到实践中时会用到的基本语法，可供你快速参考。

- 附录 A 重述了定义颜色的多种方法，包括所有预定义的颜色关键词。
- 附录 B 总结了所有的渲染服务元素、它们的属性，以及相关的样式属性。

关于本书

无论你是随便翻阅本书，还是从头到尾仔细阅读，理解如下用于提供额外信息的小指南，你可以获得更多知识。

关于示例

SVG 图像可以使用许多不同类型的软件来展示和操作，且每个程序在 SVG 代码的解析上略有不同。尤其在图形文件分发在 Web 上时，这确实是个问题；你希望人们在另一端看到的内容与你期望的效果无限接近。

因此本书中的例子在桌面最新稳定版（2015 年 7 月）的 Chrome、Firefox、Internet Explorer 以及 Safari 浏览器中都进行了测试。缺陷、漏洞以及浏览器的支持程度都在文本中有所注明。此外，还提到了微软 Edge 浏览器预期在支持程度上的更改。

几乎所有其他浏览器使用的都是某一主要开源渲染库的变体：Gecko（Firefox）、WebKit（Safari 以及 iOS 设备）或 Chromium/Blink（WebKit 的一个分支，主要为 Chrome 开发）。因此，你可以以主要浏览器的支持情况作为指南，但要注意的是并非所有软件都会同时更新。对于移动浏览器，即使某些功能在技术上是支持的，但会受到实际性能的限制。某些移动浏览器（例如 Opera Mini）会刻意限制它们支持的功能来提升性能。

SVG 还可以用在 Adobe Illustrator 和 Inkscape 等图形程序中。有很多工具，例如 Apache Batik 或 libRSVG，可以把 SVG 代码转换为其他矢量图形格式，比如 PDF 文档。这些工具都会有新的兼容性问题，这在本书中没有详细介绍。一定要小心地在所有需要使用的工具中测试！

使用代码示例

补充材料（代码示例和图形）可以从以下网址在线获取。

下载地址：https://github.com/oreillymedia/SVG_Colors_Patterns_Gradients。

在线观看地址：http://oreillymedia.github.io/SVG_Colors_Patterns_Gradients/。

本书是要帮你完成工作的。一般情况下，如果书中提供了示例代码，你可以在你的程序或文档中使用，且不用联系我们获得许可，除非你大量复制我们的代码。例如，在你的程序中使用本书中的几个代码块不需要获得我们的许可。销售或分发 O'Reilly 图书中示例的 CD-ROM 需要获得许可。引用本书及引用示例代码来回答问题不需要获得我们的许可。将本书中大量示例代码合并到你的产品文档中时需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*SVG Colors, Patterns & Gradients* by Amelia Bellamy-Royds and Kurt Cagle (O'Reilly). Copyright 2016 Amelia Bellamy-Royds and Kurt Cagle, 978-1-4919-3374-9."

如果你觉得对代码示例的使用不属于合理使用或超出了上述许可范围，请随时通过 permissions@oreilly.com 联系我们。

排版约定

本书使用的排版约定如下。

- 楷体

  表示新术语。

- 等宽字体（`constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）

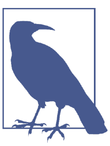
  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`Constant width italic`*）

  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标用于突出 SVG 中特别复杂的部分，或初看不太明显的简单的快捷方式。



该图标表示一般的注记和有趣的背景信息。



这样的警告信息将用于突出不同浏览器（或其他软件）之间，或 SVG 作为 XML 文件和在 HTML 页面中使用 SVG 之间的兼容性问题。

除此之外，如下样式将用于补充信息。

简介

本书中使用了两种类型的附注栏。“聚焦未来”将讨论尚未标准化的拟定功能，或还没有广泛实施的新标准。“CSS 与 SVG”将 SVG 图形效果和创建相似效果的 CSS 样式（如果有的话）进行比较。

虽然附注栏对于理解 SVG 颜色、图案和渐变并不是必要的，但在规划一个完整的 Web 项目时它们可能会添加重要的上下文。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://bit.ly/svg-colors-patterns-and-gradients

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

如果没有 O'Reilly 的编辑们——Simon St. Laurent、Meghan Blanchette 以及 Meg Foley（排名不分先后）——的耐心与坚持，这本书永远不会出版。非常感谢技术审查人员，是他们尽最大努力减少错误的数量并修改难以理解的表述，最终使本书达到出版质量要求，他们是：David Eisenberg、Dudley Storey、Robert Longson 以及 Sarah Drasner。

感谢 O'Reilly 团队，是他们使最终成书精美且专业。尤其感谢 Sanders Kleinfeld 调整 Pygmentize 语法来高亮显示 SVG 代码，感谢制作编辑 Colleen Lobner 应对了许多自定义请求。

还要感谢广泛的 SVG 开发者社区，包括那些使用 SVG 的开发人员和构建基础软件的开发人员。本书中许多高亮的提示、技巧、警告都是从其他人分享的博客文章、现场演示、问答论坛以及邮件列表等中收集的。

电子书

扫描如下二维码，即可购买本书电子书。

第 1 章

你应该知道的事

本书假设你已经了解过 SVG、网页设计，甚至有 JavaScript 编程基础。

每一种语言都有自己的一些缺陷，而这些缺陷对有些开发者来说很明确，但对其他同样优秀的开发者来说却闻所未闻。所以本章将带你一起快速回顾一些你必须知道的东西。

1.1　SVG通过代码来画图

SVG 就是一个图片文件。我们可以像使用 PNG 或 JPEG 等图片文件一样使用它，可以在可视化编辑器中创建和编辑 SVG，也可以将其作为图片嵌入到网页中。

但是 SVG 并不仅仅是图片，它是包含标记元素、文本、样式指令的结构化文档。其他图片格式是告诉计算机在屏幕上哪一点应该绘制什么颜色，而 SVG 是告诉计算机如何通过它的组成部分重组图形。这产生以下两个主要结果。

- SVG 最终在屏幕上的显示依赖于软件对 SVG 规范的支持程度。跨浏览器的兼容性往往是一个问题。
- 对 SVG 的一部分单独进行增加、删除、修改的操作是非常容易的，不用担心会改变 SVG 的其他部分。我们可以在编辑器中进行此类操作，也可以动态地在网页中制作动画或交互图形。

1.2 SVG始终是开源的

SVG 不仅仅是一组计算机编码指令，它还是人类可读的文本文件。你不仅可以在文本编辑器中编辑 SVG，甚至还可以在有语法高亮和自动补全功能的代码编辑器中编辑它。

本书的所有例子都专注于基本的 SVG 代码。当然你也可以通过可视化编辑器来画形状、选颜色、调整图形展现的其他部分。但是为了完全可控，你需要看一看编辑器实际创建的代码。

1.3 SVG是XML（有时也是HTML）

文本编辑器中的 SVG 代码看起来非常像 HTML 代码（全是尖括号和属性），但是单独的 SVG 文件通常被解析为 XML。这意味着你的 SVG 通常都可以被 XML 工具解析和操控，也意味着如果你忘记引入 XML 的命名空间或者混淆了 XML 语法的重要细节，你的网页将什么都不显示。

然而，当你直接在 HTML5 标记中插入 SVG 时，它将由 HTML 解析器处理。HTML 解析器会忽略一些错误（比如缺少结束标签或使用不带引号的属性），而在 XML 解析器（或大多数仅支持 SVG 的图形编辑器）中这将导致解析失败。它也会忽略自定义的命名空间，将不能识别的属性或标签名变成小写，甚至引起其他不能预期的变化。

1.4 SVG是可压缩的

SVG 的大部分语法的设计都是为了使其易于阅读和理解，而不是为了结构紧凑，这使得某些 SVG 文件看起来相当冗长和冗余。但是，这也使得 SVG 非常适合通过 gzip 压缩，而网页中的 SVG 都应该被压缩。通常，这将使文件大小减少一半甚至更多。在普通的文件服务器上存储压缩过的 SVG 时，通常使用 .svgz 作为扩展名。

SVG 也是容易臃肿的，这也在另一方面使它可压缩。大多数 SVG 编辑器通过给定唯一的 XML 命名空间，在 SVG 文件中添加自己的元素和属性。有些优化工具开发了在不影响最终结果的基础上把代码剥离出来的功能。在使用这些工具时，如果你自己操作了代码，配置的时候就要特别小心，优化工具有可能会删除掉你以后想要使用的属性。

1.5 图片是形状的集合

那么这所有的代码想要表现的是什么呢？当然是形状！（也可能是文本或嵌入的图像，这些我们将在后面介绍。）SVG 只有少数不同的形状元素：<rect>、<circle>、<ellipse>、<line>、<polyline>、<polygon> 以及 <path>。但是，最后三种能够以一定的精确度，定制你能想象的任何形状。特别是 <path>，我们可以通过它自己的特性来描绘曲线和线条，以此来创建形状。

1.6 图片中可以包含图片

每个 SVG 都是一张图片，但是它同时也是一个文档，文档中可以通过使用 <image> 标签来包含其他图片。嵌入的图片可以是其他 SVG 文件，也可以是 PNG 或 JPEG 等光栅图片。然而，出于安全和性能的考虑，SVG 有时用于避免外部图片（以及其他外部资源，比如样式表和字体）被下载。尤其是当 SVG 在 HTML 页面中作为嵌入图片（<img> 元素）或者背景图片被显示时，我们将不能使用外部文件。

1.7 文本也是艺术

SVG 的最后一个用途是构建文本。但是文本不是图形的替代，构成文本的字母将会像其他类型的矢量图形一样被处理。尤为重要的是，文本可以使用与矢量形状完全相同的样式属性来绘制。

1.8 艺术源于数学

所有矢量图形（形状或文本）的核心是，使用每个元素的浏览器 SVG 渲染函数中的数学参数（XML 属性），控制最终效果。SVG 中最基本的数学概念是坐标系，用于确定图形中每一个点的位置。我们可以通过设置 viewBox 属性来设置初始坐标系，然后通过坐标系的转换来移动、拉伸、旋转、倾斜特定元素。

1.9 SVG是无数canvas的有限集

在计算机精度允许的范围内，可以给矢量形状添加任意多个坐标。不过最终显示的是 viewBox 属性建立的特定范围坐标内的形状。通过给 preserveAspectRatio 设置不同的值来控制在宽高比不匹配时，如何把坐标

的范围缩放到视图区域（viewport）。

嵌套 `<svg>` 元素或复用 `<symbol>` 元素可以创建嵌套的视口，它们除了提供控制宽高比的区域外，还定义了确定子元素百分比值的依据。其他元素还可以使用 `viewBox` 属性来创建一个伸缩到合适大小的效果（我们将在第 11 章的 `<pattern>` 元素中学习），而不用重新去定义百分比。

1.10 SVG是有结构的

SVG 的结构包括渲染到屏幕上的基本形状、文本和图片以及用来定义几何图形的属性。我们可以按照逻辑来把元素进行分组，以此来创建更加丰富的 SVG。我们可以给不同的组设置不同的样式或者转换其坐标系，还可以给它们添加可访问名称和描述来解释图形到底表示什么。

1.11 SVG是有样式的

SVG 图形可以仅仅由所有的样式信息都通过属性来设置的 XML 元素组成。当然，这些和表现相关的样式也可以通过 CSS 规则来设定，比如通过 class 或者元素类型来控制。还可以使用媒体属性或瞬时状态等有条件的 CSS 样式，比如 `:hover` 和 `:focus`。

严格分开几何结构（XML 属性）和表现样式（表现属性或者 CSS 样式规则）是有些武断的。随着 SVG 的发展，这两者之间的界限将越来越模糊。SVG 2 的规范草案把许多布局属性升级为表现属性。这便可以通过灵活的 CSS 语法来提供以下特性：通过类给相似元素设置一个差不多的尺寸，然后再通过 CSS 伪类或媒体查询来修改具体的尺寸或布局。

1.12 所有好用的标记都基于伟大的DOM

浏览器会把 SVG 标记和样式转换成一个文档对象模型（document object model，DOM）。DOM 可以通过 JavaScript 进行操作。针对 XML 内容的 DOM 的所有核心方法同样适用，所以我们可以创建和重排元素，获取和设置属性的值，查询计算后的样式的值。

SVG 规范还定义了 SVG DOM 元素特有的属性和方法，这使得我们可以更简单地从数学角度操作几何图形。浏览器对 SVG DOM 的支持可能不尽如人意，但一些方法——比如确定一条曲线的长度——在 SVG 的设计中是不可或缺的。

1.13 SVG是可移动的

在支持脚本的动态 SVG 查看器（比如 Web 浏览器）中，可以使用脚本来创建动画和可交互的图形。SVG 还支持交互的声明方式，你可以定义整个交互的范围，浏览器通过自己的优化方式来应用它。这有两种实现方式。

- 借鉴 SMIL（Synchronized Multimedia Integration Language，同步多媒体集成语言）的语法，在标记中使用动画元素。
- 在表现样式中使用 CSS 动画和过渡。

在编写本书的时候，脚本动画已经得到所有 Web 浏览器的支持，但可能因为 SVG 的某些用途而受到阻碍。声明式动画（SMIL 和 CSS）在大多数浏览器中也得到了支持，但并不是所有（IE 浏览器显然是个例外）。此外，浏览器也开始实现新的 Web 动画 API，这将使得脚本可以像声明式动画那样定义和触发独立运行的动画。

1.14 SVG在发展变化

在你与 SVG 交互时，不仅 SVG 图形可以单独改变，SVG 的定义也可以改变。目前（撰写本书时）的既定标准是 SVG 1.1，但具有新特性和更明确地定义已有特性的 SVG 2 规范正在制定中。此外，由于 SVG 使用了 CSS 和 JavaScript，并且由于它很大程度地集成在 HTML 中，这些语言的变化也将影响 SVG。

第 2 章

画家模型

如果我让你画一个黄色的圆，然后给它加一个蓝色的轮廓，它看起来会和画一个蓝色的圆并用黄色填充的效果一样吗？

如果我让你画一个红色的五边形和一个绿色的正方形，且它们的中心点是页面中的同一点，图片的大部分会是红色还是绿色？

在用手画图时是没有固定规则的。如果有人给了你模棱两可的指示，你通常会向他寻求解释。但是当你向计算机发出指令时，它只能按着你的要求执行，所以你必须保证能准确地表达你的意思。

即使你使用 SVG 很长时间了（我们假设你至少使用过一段时间），也很可能没有深入思考过计算机是如何把 SVG 代码转换成屏幕上的彩色图案的。如果你的确要画许多这样的彩色图案，就需要知道你的指令是如何被解析的。

本章将讨论 SVG *渲染模型*的基本原理，也就是计算机解析 SVG 标记和样式来生成图画的过程，还将回顾定义 SVG 图形和文本绘制方式的基本属性——`fill` 和 `stroke`。本书其余部分的内容可以概括为定义 `fill` 或 `stroke` 属性值的不同方式。

SVG 的渲染模型被称为*画家模型*。就像在墙上涂层，上层的内容会遮盖下层的内容。SVG 规范定义了哪些内容放在其他内容之上。本章还会讨论 `z-index` 和 `paint-order` 这两个允许你改变渲染规则的属性。这两个属性由 SVG 2 引入，并且刚刚开始在一些 Web 浏览器中得到支持。所以我们也会介绍如何用 SVG 1.1 的代码来实现相同的效果。

2.1 使用fill属性进行填充

SVG 代码中的基本元素和属性可以精确地定义几何形状。例如创建一个 1 平方英寸的正方形，它的左上角就是坐标系的原点，代码如下所示：

```
<rect width="1in" height="1in" />
```

创建一个直径为 10 厘米的圆，并把它放置在坐标系的中心，代码如下：

```
<circle cx="50%" cy="50%" r="5cm" />
```

> 本书不准备花费大量的时间去讲解你画的图形的几何结构。但需要注意的是，SVG 通常会被缩放，所以英尺、厘米等长度单位不一定符合真实世界的距离。它会受到显示器的分辨率、浏览器的缩放程度，当然还有 SVG 元素上添加的 `viewBox` 或 `transform` 等属性的影响。
>
> 缩放对所有单元产生的影响是相同的。通常每英寸包含的厘米数（2.54）和你用尺子量是一样的。除了最古老的浏览器，在所有浏览器中，`1in` 也总是等于 `96px`（CSS 像素单位），同样也等于 96 个 SVG 无单位用户坐标值。SVG 无单位坐标通常可以和 `px` 等价使用。其他的 SVG 软件同样也遵循这项在 CSS Values and Units Module Level 3 中建立的标准。

如果在你的 SVG 中仅仅包含一个圆或一个长方形的标记（或其他任何形状或文本）而没有其他的样式信息，它将在你定义的尺寸内显示一个纯黑色的区域。这是因为 `fill` 属性的默认值是纯黑色。

`fill` 属性告诉 SVG 渲染软件如何操作几何形状。对于屏幕上的每一个像素（可以对比纸上的每一个墨斑），软件都可以确定该点是不是在形状之内。如果该点在形状之内，软件就会填充 `fill` 的值，然后确定下一步怎么做。

在简单的例子（如默认黑色）中，填充值是一种颜色，且形状内所有的点都用该颜色替换。在其他情况中，填充值可能是一个查找更加复杂的绘画代码的指令。该指令是通过引用一个 SVG 元素的 `id` 的 URL 来表示的（一种渲染服务，我们将在第 5 章深入讨论）。

> 如果你不想让软件填充形状，可以把 `fill` 属性值设置为 `none`。

fill 属性（以及 stroke 属性）的最后一个可以设置的值是 currentColor 关键词。这一关键词通常被估算为给定元素的 CSS color 属性的当前值。color 属性本身对 SVG 没有直接的影响，但是结合 currentColor，它将有两个主要用途。

- 使内联的 SVG 图标与它周围的 HTML 文本颜色协调。color 属性的主要用途是设置 CSS 样式文本的颜色。因此，使用 currentColor 值的内联 SVG 图形会继承周围 HTML 标记的文本颜色。
- 为重复使用的内容提供一个间接继承的样式值。使用 <use> 元素复制的 SVG 图形可以从使用它的上下文中继承 fill 和 stroke 等样式。给重复使用的图形中的重要属性使用 currentColor，这样可以通过改变 <use> 元素上的 color 值来分开操作复用图形的 fill 和 stroke 的值。

默认情况下，fill 属性被渲染为纯色且不透明（除非在渲染服务中有不同的指令）。可以通过给 fill-opacity 属性设定值来调整不透明度，它接受一个小数作为值：0 到 1 之间的值会导致填充值和背景色混合起来渲染图形，值为 1（默认值）时表示不透明，值为 0 时的效果相当于设置 fill 属性的值为 none。我们将在第 4 章中讨论不透明度。

当你不能确定图形的某一部分是在图形之内还是之外时，fill-rule 属性可以给计算机发送精确的指令。它会影响内部有洞的 <path> 元素以及路径、多边形和纵横交错的折线。

fill-rule 属性有两个可选值。

- evenodd 值的每一条边缘线都分隔开了图形的内部和外部。
- nonzero 值（默认值）是当你从头到尾沿着一个方向画交叉的边线时企图得到“更多的内部空间”，并且只有你在图形内部沿着相反方向绘画来撤销它们时才会返回图形外部。

例 2-1 画的是一个纵横交错的 <polygon>，第一个使用默认的 nonzero 填充规则，另一个使用 evenodd 填充规则。图 2-1 显示了渲染结果。图形的边上有细细的描边，所以即使你填充了边线的两侧，也可以看到形状。

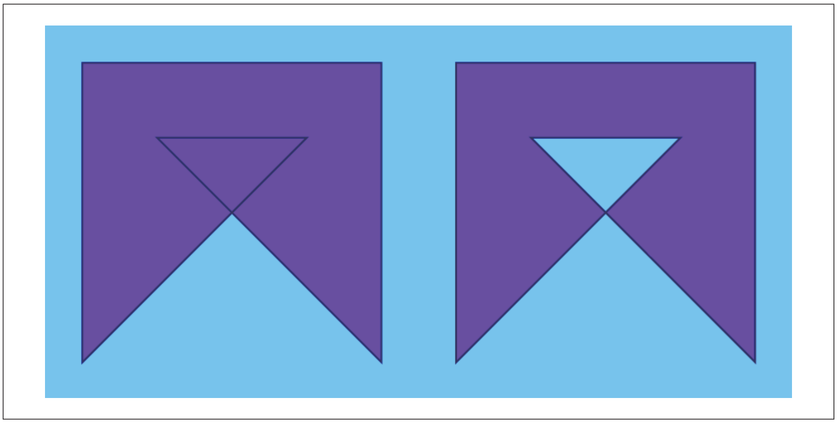


图 2-1：使用 nonzero 填充规则的多边形（左）和使用 evenodd 填充规则的多边形（右）

例 2-1　使用 fill-rule 属性改变填充区域

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     viewBox="0 0 400 200" width="4in" height="2in">
    <title xml:lang="en">Fill-rule comparison</title>
    <rect fill="lightSkyBlue" height="100%" width="100%" />       ❶

    <polygon id="p"
             fill="blueViolet" stroke="navy"
             points="20,180 20,20 180,20 180,180 60,60 140,60" /> ❷
    <use xlink:href="#p" x="50%" fill-rule="evenodd" />           ❸
</svg>
```

❶ 最初的 <svg> 元素建立了坐标系，并且设置了图形绘画的默认尺寸。<rect> 元素增加了一个纯色背景。在这段简单的 SVG 代码中，我们直接在表现属性上设置样式。

❷ 最基本的拥有 fill 和 stroke 样式的多边形，它的 fill-rule 属性将继承默认的 nonzero 值。

❸ 一个水平移动 SVG 一半宽度的同一个多边形的副本。复制的多边形将继承在 <use> 元素上设置的 fill-rule 的值 evenodd。

无论边线或者子路径相互交叉多少次，每个点不是在图形内部就是在图形外部。每个区域不会因为它在两个不同的子路径内而被绘画两次。当使用纯色填充时这点差别看似没有多大意义，但使用部分透明来填充时它就变得非常重要。

聚焦未来

未来的填充

本节讨论的 fill 属性都是基于当前已经发布的稳定的 SVG 1.1 规范。在制定中的 SVG 2 规范将提供更加灵活的方式去填充图形，尤其值得关注的是允许单个图形拥有多个填充层。这些被提及的特性将在本书其他部分作更详细的讨论，同样也是在这样的“聚焦未来”附注栏中。

SVG 中的每个图形和文本都可以被填充，且默认是填充的。这包括不闭合的 <path> 元素和 <polyline> 元素，它们可以定义一个结束点不与起始点相连的图形。这些图形会创建一个结束点和起始点用直线相连的填充区域。如果在结束时与其他的边有交叉，fill-rule 属性就会计算并应用。

> <path> 中没有闭合的片段是通过连接到子路径的初始点来闭合的：最后的点是通过一个 move-to 命令创建的。

即使是一个 <line> 元素严格来说也是默认填充的：因为连接终点与起始点的返回线与原线完全重合，所以最后的形状不包括任何区域。形状内是没有点的，所以没有点被填充值影响。如果你想看到它，就得给它加 stroke。

2.2　使用stroke属性描边

在计算机图形中，给形状描边的意思是指沿着它的边画一条线。不同的程序对描边的含义会有不同的解析。

在 SVG 中（目前如此），描边的实现方式是沿着主形状的边线向内和向外延伸出辅助形状。该描边区域使用与填充主形状相同的方式来渲染：软件会依次扫描并确定某个点是否在描边区域的内部。如果在，软件就会使用 stroke 属性设置的渲染指令来设置颜色。

> 描边形状的每个部分都只会绘制一次，无论形状中是否存在不同边重叠或交叉的部分。

`stroke` 的默认值是 `none`，即不渲染描边区域。就像 `fill` 属性一样，它的值还可以是色值或渲染服务元素的引用。

就像 `fill-opacity` 属性会改变 `fill` 的效果一样，描边同样也有 `stroke-opacity` 属性来改变它的效果。第 4 章会详细讨论 `fill-opacity` 和 `stroke-opacity` 属性。

还有许多与描边相关的属性，本书不准备用大量篇幅去讨论它们。但需要知道，它们可以控制描边区域的几何形状。以下是这些属性的大致介绍。

`stroke-width`
: 描边宽度，即描边的粗细。其值可以是长度值、用户单位数或者坐标系宽和高的加权百分比。在 SVG 1.1 中，描边区域通常以形状的边为中心，所以描边的一半宽度在形状之内，一半在形状之外。

`stroke-linecap`
: 该属性用来给未闭合的路径或线条设置描边样式。其默认值 `butt` 会紧密修剪描边并且与端点垂直。其他选项（`round` 和 `square`）会以特定形状（即分别以半圆形和方形）使用一半的描边宽度来延伸描边。

`stroke-linejoin`
: 该属性用于指定在形状中拐角的描边样式。其默认值 `miter` 在直线上延伸描边，直到两条边在某一点相汇。其他可选值是 `round`（使用圆弧来连接两条描边）和 `bevel`（使用一根额外的直线连接两条描边）。

`stroke-miterlimit`
: 延伸斜接线可以超出形状边线的最大距离，是描边宽度的倍数（默认是宽度的四倍）。如果描边在这个距离之内没有汇合，则使用 `stroke-linejoin` 值为 `bevel` 时的效果。

`stroke-dasharray`
: 定义给形状间断描边时的距离模式（线和间隔）。其默认值 `none` 会给整个形状添加连续的描边。每一条线的端点都受 `stroke-linecap` 值的影响。

`stroke-dashoffset`
: 定义间断描边时起始偏移的距离。默认值是 0。

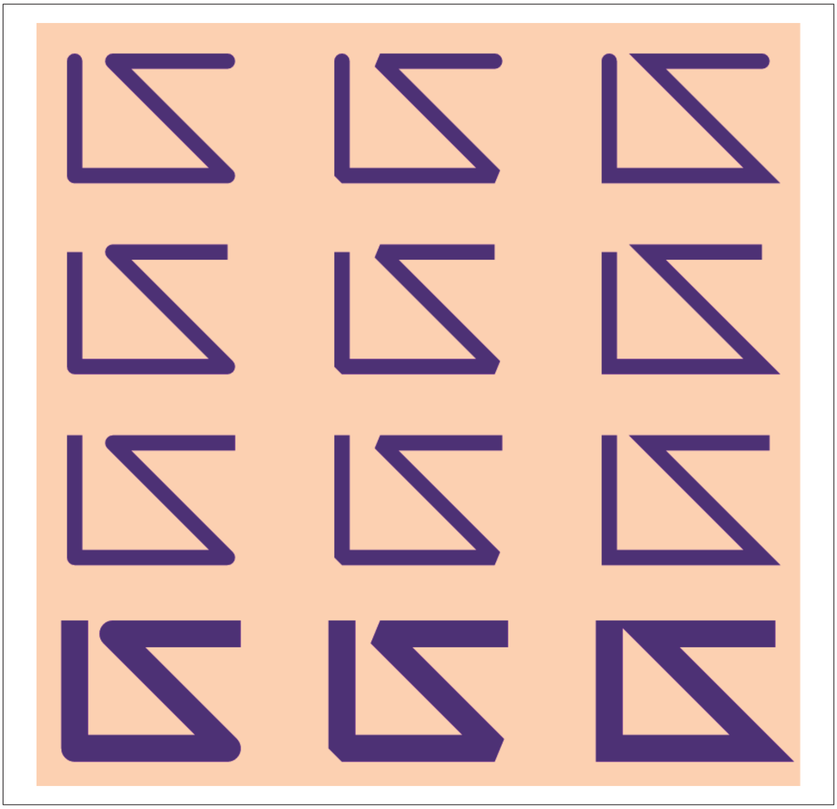


图 2-2：描边形状变化的折线。从左到右，stroke-linejoin 属性值分别为 round、bevel 和 miter。从上到下，stroke-linecap 属性值分别为 round、butt 和 square。最后一行：stroke-linecap 属性值为 square 且 stroke-width 更粗

例 2-2 在同样的折线上混合搭配使用不同的 stroke-linejoin 和 stroke-linecap 属性值，生成的结果如图 2-2 所示。

例 2-2　控制描边区域的几何形状

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     viewBox="0 0 400 400" width="4in" height="4in"
     xml:lang="en">
    <title>Stroke join and cap styles compared</title>
    <style type="text/css">
        .backdrop {                                    ❶
```

```
        fill: peachPuff;
    }
    .shapes {
        fill: none;
        stroke: indigo;
        stroke-width: 8px;
    }
    .join-round {
        stroke-linejoin: round;
    }
    .join-bevel {
        stroke-linejoin: bevel;
    }
    .join-miter {
        stroke-linejoin: miter;
        stroke-miterlimit: 10;
    }
    .cap-round {
        stroke-linecap: round;
    }
    .cap-butt {
        stroke-linecap: butt;
    }
    .cap-square {
        stroke-linecap: square;
    }
    .wider {
        stroke-width: 14px;
    }
</style>
<defs>
    <polyline id="p2"
        points="20,20 20,80 100,80 40,20 100,20" />    ❷
</defs>
<rect class="backdrop" height="100%" width="100%" />

<g class="shapes">
    <g class="cap-round">
        <title>Round line caps</title>
        <g id="row">                                    ❸
            <use xlink:href="#p2" x="0" class="join-round">
                <title>Round line joins</title>
            </use>
            <use xlink:href="#p2" x="35%" class="join-bevel">
                <title>Beveled line joins</title>
            </use>
            <use xlink:href="#p2" x="70%" class="join-miter">
                <title>Mitered line joins</title>
```

```
                </use>
            </g>
        </g>
        <g class="cap-butt">                                    ❹
            <title>Butt (cropped) line caps</title>
            <use xlink:href="#row" y="25%" />
        </g>
        <g class="cap-square">
            <title>Square line caps</title>
            <use xlink:href="#row" y="50%" />
        </g>
        <g class="cap-square wider">                            ❺
            <title>Square line caps with a wider stroke</title>
            <use xlink:href="#row" y="75%" />
        </g>
    </g>
</svg>
```

❶ 在这个更加复杂的例子中，使用 CSS 规则的样式块根据它们的类来给元素设置 `fill` 和 `stroke` 属性值。

❷ 在 `<defs>` 部分里预先定义基本的 `<polyline>` 形状。

❸ 复制三份折线并将其排列到对比网格的一行中。每一个都有不同的 `stroke-linejoin` 样式。

❹ 复制一整行并垂直移动，每次都分配一个新的 `stroke-linecap` 样式。

❺ 最后一行继承了不同的 `stroke-width` 值。

与填充不同的是，描边区域会受到未闭合路径的影响。给 `<polyline>` 元素和设置相同点的 `<polygon>` 元素描边看起来是不一样的。使用 `Z` 命令闭合的 `<path>` 元素（或其子路径）会有一条线连接终点和起始点，而开放的子路径的起始点和终点都是端点。

聚焦未来

下一代描边

描边属性在 SVG 2 中也将有所改变，它将允许每个形状有多个描边层。许多关于更好地控制描边几何形状的提案也正在被考虑，它们将在一个单独的 SVG 描边模块上进行开发。

2.3　层叠描边和填充

当一个图形同时拥有填充属性和描边属性时，描边区域和填充区域会有一部分重合的地方，因此重合部分会有两种特定的颜色。在所有的 SVG 中，画家模型都适用：如果两种颜色都是不透明的，则上层的颜色将会替换下层的颜色。

但是哪一层是在“上面”的呢？

默认情况下，描边是渲染在填充层之上的。这意味着你通常可以看到整个描边宽度，也意味着如果描边是半透明的话，将显示出两种颜色。填充的颜色将会出现在描边区域内部一半之下而不是外部一半之下。

> 描边标记——自定义形状拐角显示的标志——是在填充和描边之后渲染的，按照路径从头到尾的顺序。

在 SVG 1.1 中，在把描边区域放到填充区域下的唯一方式是分成两个形状：一个仅仅描边，然后把相同的图形复制到同一位置（使用 <use> 元素）并仅填充不描边。

```
<g stroke="blue" fill="red">
    <g fill="none">
        <path id="shape" d="..." />
    </g>
    <use xlink:href="#shape" stroke="none" />
</g>
```

前面的代码片段使用了大量继承的样式。<path> 本身没有设置任何的 fill 和 stroke 属性，而是从它的外层继承的。总的 stroke 和 fill 属性设置在外层的 <g> 元素上，之后在嵌套的组和 <use> 元素上分别抵消了 fill 属性和 stroke 属性。

SVG 2 中引入了 paint-order 属性，使得这样的效果更容易实现。它使用空格分隔的关键词（stroke、fill 以及 markers）列表来指示图形的各部分被渲染的顺序。所以可以用单个元素实现相同的效果：

```
<path id="shape" d="..." stroke="blue" fill="red"
      paint-order="stroke fill" />
```

你在 paint-order 属性中没有定义的渲染层将会在之后渲染（本例中的

markers），并按照它们本来的顺序渲染。这意味着如果你想调换 fill 和 stroke 的顺序，仅需要定义 stroke 就可以了。

```
<path id="shape" d="..." stroke="blue" fill="red"
      paint-order="stroke" />
```

stroke 将会最先被渲染，然后是 fill，最后是 markers。整个填充区域将始终可见，即使是与描边重叠的地方。

paint-order 的默认值（等于 fill stroke markers）可以用 normal 关键词显式地设置。



在编写本书的时候，paint-order 属性在最新的 Firefox（从 31 版本开始）、Blink（从 35 版本的 Chromium 开始）以及 WebKit（从 2014 年 3 月开始）中得到支持。IE/Edge 以及其他老版本的浏览器使用的是默认的渲染顺序。

控制渲染顺序的能力在文本中尤为重要。SVG 中的文本可以像形状一样通过描边来创建轮廓的效果。然而，即使最细的描边也会遮盖字母的细节。

通过在描边之上渲染填充区域（在颜色不同的时候），你可以加强字母的形状并恢复它的易读性。例 2-3 使用 paint-order 和一个很粗的描边给标题文本加了一个清晰的轮廓。图 2-3 是在支持的浏览器中显示的结果。



图 2-3：在填充之下描边的 Outlined 文本

例 2-3　描边没有遮住文本的细节

```
<svg xmlns="http://www.w3.org/2000/svg"
     viewBox="0 0 400 80" width="4in" height="0.8in"
     xml:lang="en">
    <title>Outlined text, using paint-order</title>
    <rect fill="navy" height="100%" width="100%" />
    <text x="50%" y="70"
          text-anchor="middle"
          font-size="80"
          font-family="sans-serif"
```

```
          fill="mediumBlue"
          stroke="gold"
          stroke-width="7"
          paint-order="stroke"
          >Outlined</text>
</svg>
```

如果完全依赖 paint-order 来实现这种效果，那你的文本在不支持的浏览器中将表现得一塌糊涂，如图 2-4 所示。所以应该要有备选策略。



图 2-4：使用默认渲染顺序描边的 Outlined 文本

一种解决方式是使用 CSS 的 @supports 条件规则来控制只有在支持 paint-order 属性时才添加轮廓效果。其他情况下，如果没有达到预期的效果，就使用不同的样式来提供清晰的文本。

例 2-4 是例 2-3 代码的一个变形。样式从表现属性上移到了 style 代码块中，这样就可以使用 CSS 的条件规则了。基本的样式是在不能控制渲染顺序的时候提供更细的描边，在 @supports 块中使用粗的描边和 paint-order 属性来覆盖基本样式。

例 2-4　使用 paint-order 属性前先测试是否支持

```
<svg xmlns="http://www.w3.org/2000/svg"
     viewBox="0 0 400 80" width="4in" height="0.8in"
     xml:lang="en">
    <title>Using @supports to adjust paint-order effects</title>
    <style type="text/css">
        .outlined {
          text-anchor: middle;
          font-size: 80px;
          font-family: sans-serif;
          fill: mediumBlue;
          stroke: gold;
          /* fallback */
          stroke-width: 3;
        }
```

```
            @supports (paint-order: stroke) {
                .outlined {
                  stroke-width: 7;
                  paint-order: stroke;
                }
            }
      </style>
      <rect fill="navy" height="100%" width="100%" />
      <text x="50%" y="70" class="outlined"
            >Outlined</text>
    </svg>
```

在支持 paint-order 的浏览器（目前这些浏览器也都支持 @supports 规则）中的运行结果如图 2-3 所示。修改之后的代码在其他浏览器中的运行结果如图 2-5 所示。



图 2-5：当不支持 paint-order 时给文本添加更细的描边



图 2-3 和图 2-5 之间 stroke-width 的值被裁掉一半还多。但是图 2-5 中描边仅仅略微窄一点，这是因为描边内部的一半在填充之上是可见的。

如果你不能接受使用 @supports 来改变展现规则，唯一的替代方案就是复制两个相同的元素，一个用来描边，一个用来填充。根据你使用 SVG 的方式以及对样式的控制程度，必要时可以使用脚本来控制转换。由于 paint-order 是一个新的 CSS 属性，不支持它的浏览器不会把它添加到每个元素的 style 属性上。因此，你可以检测浏览器是否支持，并在需要时生成额外的 <use> 元素。

例 2-5 中提供了一段脚本样例，它通过类名来鉴定元素并在需要的时候执行一系列操作。

例 2-5　使用多个元素来模拟 paint-order

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     viewBox="0 0 400 80" width="4in" height="0.8in"
     xml:lang="en">
    <title>Faking paint-order with JavaScript</title>
    <style type="text/css">
        .outlined {
          text-anchor: middle;
          font-size: 80px;
          font-family: sans-serif;
          fill: mediumBlue;
          stroke: gold;
          stroke-width: 7;
          paint-order: stroke;
        }
    </style>
    <rect fill="navy" height="100%" width="100%" />
    <text x="50%" y="70" class="outlined"
          >Outlined</text>
    <script><![CDATA[
(function(){
    var NS = {svg: "http://www.w3.org/2000/svg",
              xlink: "http://www.w3.org/1999/xlink"
             };
    var index = 10000;

    var t = document.getElementsByClassName("outlined"); ❶
    if ( t &&
        (t[0].style["paint-order"] === undefined )){        ❷
       Array.prototype.forEach.call(t, fakeOutline);        ❸
    }

    function fakeOutline(el){
        el.id = el.id || "el-" + index++;                    ❹

        var g1 = document.createElementNS(NS.svg, "g");     ❺
        g1.setAttribute("class", el.getAttribute("class") );
        el.removeAttribute("class");
        el.parentNode.insertBefore(g1, el);

        var g2 = document.createElementNS(NS.svg, "g");     ❻
        g2.style["fill"] = "none";
        g2.insertBefore(el, null);
        g1.insertBefore(g2, null);

        var u = document.createElementNS(NS.svg, "use");    ❼
        u.setAttributeNS(NS.xlink, "href", "#" + el.id);
        u.style["stroke"] = "none";
```

```
            g1.insertBefore(u, null);
        }
    })();
    ]]> </script>
    </svg>
```

❶ 为了方便在脚本中访问，用一个特定的类名 "outlined" 来标示要修改的元素。

❷ 可以检查任何元素（这里是指第一个被选中的元素）的 style 属性，以确定它是否支持 paint-order 属性。使用严格相等测试（===）来区别空值（元素上没有设置内联样式）和 undefined 值（属性名无法识别）。

❸ 如果需要回退，就会在每一个有 "outlined" 类名的元素上调用 fakeOutline() 方法。数组方法 forEach() 用于根据需要多次调用该方法。由于 getElementsByClassName() 返回的列表不是一个真正的 JavaScript 数组，所以不能使用 t.forEach(fakeOutline)。取而代之的是从数组原型中提取出 forEach() 方法，并使用自己的 call() 方法来调用。

❹ fakeOutline() 方法将会使用 <use> 元素来复制轮廓元素，所以它需要有一个有效的 id。如果它本身没有，我们会给它添加一个唯一的随机值。

❺ 把元素的类复制到组元素上并删除元素本身上的类，并用组元素替换原元素。当然，这要求所有填充和描边样式都定义到类上，而不是通过标签名定义或通过表现属性添加。insertBefore() 方法用于保证新的组元素在 DOM 树中的位置和它替换的元素相同。

❻ 嵌套的组元素是用于保持原元素的样式，防止它继承填充样式。

❼ 最后使用 <use> 来复制元素，但是要取消描边样式而只继承填充样式。把它插入到主要组元素的最后（在 null 之前），这样它就会在没有填充样式版本的元素之上绘制。

脚本运行（在不支持 paint-order 的浏览器上）的结果如图 2-6 所示。虽然它看起来和图 2-3 一样，但是它底层的 DOM 结构更加复杂。



图 2-6：复制文本来模拟先描边的渲染顺序

如你所见，使用脚本来实现这样一个简单的效果是很麻烦的。而创建一个

通用的回退脚本——一个完整的属性降级方案——更加困难，因为你需要考虑样式属性应用在元素上的所有不同方式。实际上，你需要重建 CSS 解析器的工作，识别所有的样式规则并丢弃无效的规则。

更多时候，如果最终的效果在所有的浏览器上都必不可少，则在你的标记中创建层级的描边和填充对象，直接创建之前由脚本生成的结构，这样更容易。

```
<g class="outlined">
    <g style="fill: none;">
        <text id="el-10000" x="50%" y="70">Outlined</text>
    </g>
    <use style="stroke: none;" xlink:href="#el-10000" />
</g>
```

无论是硬编码标记还是由脚本动态生成标记，在文档中其他活跃的脚本中都必须考虑复杂的 DOM 结构。

聚焦未来

使用 z-index 来控制排序

当不同的形状（或其他内容）重叠的时候，画家模型同样有效：形状是一个一个被渲染的，最后渲染的形状将会在顶部。

SVG 文档中的层级是按照代码中元素定义的顺序排列的：形状、文本和图片都会按照标记定义的确切顺序来分层。在 SVG 1.1 中，改变元素渲染顺序的唯一方式是改变元素在 DOM 中的顺序。

这样做有两个主要问题。

- 它会迫使你打破内容在逻辑上的分组。例如，如果不使用 `<g>` 元素来把文本和它描述的图形分组，你通常需要把文本标记移到文件的最后，这样才不会被其他形状遮挡。
- 你不能通过使用 SMIL 或 CSS 动画来改变看到的层级。你必须通过 JavaScript 来操作 DOM，这可能会带来性能问题或中断用户输入的焦点。

相比之下，CSS（从第二版开始）使用了 `z-index` 属性来布局。在相同的 CSS 布局层叠上下文中的重叠元素（由于固定或相对定位或者负的外边距导致）从下至上是根据 `z-index` 属性的值来排序的。确切的属性值并不重要，重要的是排列的顺序。

SVG 2 中采用 `z-index` 来重排 SVG 的层级。它的默认值是 0，我们可以给

它设置一个正值来把该元素放到其他图形前面，或者设置负值来把该元素放到后面。

> 本书写作时，还没有主流的浏览器实现 SVG 元素的 `z-index` 栈。

当某些样式应用在父元素上时，`z-index` 重排元素的能力会受到限制。滤镜、遮罩以及不透明度的值小于 1，都会导致子内容被压入一个单独的栈中，它们将作为一个整体来布局。

> 不同于 CSS 布局，在 SVG 中二维坐标系的转换不会创建一个新的层叠上下文。这说明实际上变换是 SVG 布局中很正常的一部分。

使用 JavaScript 来取代 `z-index` 的功能需要一个复杂的 polyfill 库，它需要扫描所有样式表并计算每个元素最终的层叠值。导致事情更加复杂的是：因为大多数浏览器支持 CSS 布局控制的 `z-index` 属性，所以你不能使用 `@supports` 或简单的 JavaScript 检查来确定它是否支持 SVG 的 `z-index`。

即使有这样可运行的脚本，你也无法取代 `z-index` 最重要的好处：将 DOM 的逻辑组织结构与渲染顺序分离。你的标记可能是按照逻辑顺序写的，但是如果你使用脚本把它重排，乱序的版本将被辅助工具（比如屏幕阅读器）使用，或在复制粘贴文本时使用。

遗憾的是，目前最好的做法依然是按照你想要的元素渲染顺序来组织代码。对于屏幕阅读器，你可以使用 ARIA 属性来指定逻辑分组以及元素的顺序：用 `aria-owns` 来创建一个虚拟的父子关系，用 `aria-flowto` 来定义阅读顺序。动态改变交互元素的渲染顺序还是很可能导致用户输入焦点的问题。

2.4 使用渲染提示属性

样式属性的最后一课有助于你控制浏览器如何把渲染数据运用在图形上。在此之后，本书的其他部分将重点关注你在这些形状中画什么。

这些最后的属性被认为是你（SVG 的作者）给浏览器或其他把代码转换成有色像素的软件的提示。当浏览器必须以某种方式牺牲性能或展现时，它

们将告诉浏览器或软件哪个属性是你认为最重要的。这样属性或多或少会有一致的影响。

`shape-rendering`

浏览器在屏幕分辨率的限制范围内应该如何调整形状的边缘。它有四个可选的值。

- `auto`。默认值。这告诉浏览器选择最佳的优化方式。
- `optimizeSpeed`。这告诉浏览器快速渲染是最重要的特性（可能因为图形正在执行动画），图形的边缘可能不会被精确地绘制。不过，细微的变化可能因浏览器而不同，但大部分情况下它和 `auto` 的效果是相同的。
- `crispEdges`。这告诉浏览器应该把填充和描边区域边缘的对比度最大化，这通常意味着边缘会在最近的像素的边界形成*锯齿*，而不是对边缘像素部分着色（*反锯齿*）。对于垂直线或水平线，这将创建一个锐利清晰的图像，但是对于曲线或斜线，结果往往不太令人满意。
- `geometricPrecision`。这告诉浏览器应该尽可能精确地绘制形状，如果需要可能会使用反锯齿模式。

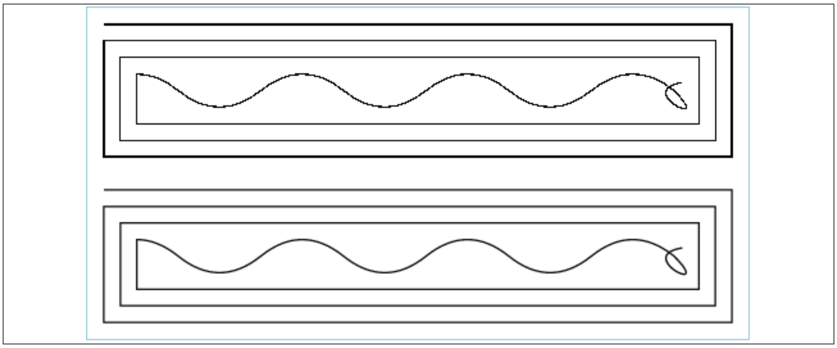


图 2-7：`shape-rendering` 属性在一条描边很细的路径上的效果：`crispEdges`（上）和 `geometricPrecision`（下）

图 2-7 对比了 `crispEdges` 和 `geometricPrecision` 属性值应用在直线和曲线上的效果。每个图形都是由一条描边宽度为 1 像素的路径组成的。值为 `crispEdges` 时，直线的边缘是锐利且干净利落的，但不均匀：由于线条的精确位置不同，导致描边的宽度在一个或两个全屏幕像素上下浮动（最窄的时候可能什么都没有），曲线会被分割为尖锐的一小段一小段。相比之下，使用 `geometricPrecision` 绘制的时候，路径上的每一个点会分配到等

量的颜色，如果需要的话，颜色还可能占据多个设备像素来进行模糊处理。

`text-rendering`

浏览器应该如何调整文本中字母的形状以及位置。它也有四个可选值。

- `auto`（默认值）。
- `optimizeSpeed`，在大多数浏览器中效果和 `auto` 相同。对于较大的文本，可能会关闭文本布局调整（对于大于 20 像素的文本，Firefox 默认使用易读性调整）。
- `optimizeLegibility`，这告诉浏览器应该尽可能地调整单个字母的渲染以及文本字符串的布局来使其更易于阅读。实践中，一些浏览器把它当作扩大字母间距以及字体文件中指定的不必要连字的暗示。
- `geometricPrecision`，这告诉浏览器应该把字母当作几何形状来精确地绘制，而不会根据基于分辨率的字体提示来调整。

在 SVG 1.1 中，没有明确定义 `text-rendering` 的确切影响。它不是一致地为 SVG 文本而实现的，但是该属性被一些浏览器用于控制非 SVG 内容的字距和连字。

不同选项值的确切影响可能在未来的规范（SVG 2 或 CSS 模块）中加以说明。`font-variant-ligatures` 和 `font-kerning` 等属性的引入应该有助于从渲染质量上区分这些特征（虽然使用 `optimizeSpeed` 渲染可能依然会导致这些设置被忽略）。图 2-8 显示了在 Windows 电脑上在 Firefox 39 中给常见的系统字体设置不同值的效果，`optimizeSpeed` 和 `optimizeLegibility` 的效果看起来一样，但在小号字体中明显与 `geometricPrecision` 渲染的结果不同。

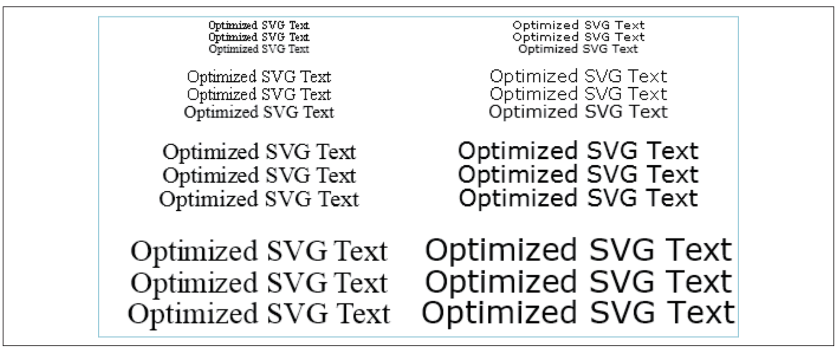


图 2-8：给不同字号的 Times New Roman（左）和 Verdana-system fonts（右）两种字体设置 `text-rendering` 属性值的效果。从上到下依次为 `optimizeSpeed`、`optimizeLegibility`、`geometricPrecision`

`color-rendering`

浏览器在计算颜色时应该精确到什么程度，尤其在使用混合元素和产生渐变时。标准的提示关键词有如下几个选项：

- `auto`
- `optimizeSpeed`
- `optimizeQuality`

浏览器目前不会响应该属性设置的任何行为。

`image-rendering`

当图片显示的大小和图片文件本身定义的像素大小不完全相同时，浏览器应该如何计算来展现光栅图片。在 SVG 1.1 中，可选的标准值有 `auto`、`optimizeSpeed` 和 `optimizeQuality`。然而在实践中，很明显，对于创建一个“高质量”缩放图片的方法并不总是一致的。在图片包含尖锐的边缘时，处理相片的最佳算法可以创建毛玻璃效果。

在 CSS Image Values and Replaced Content Module Level 3 中已经接受了 `image-rendering` 属性，但废弃了 `optimizeSpeed` 和 `optimizeQuality` 两个属性值。`optimizeSpeed` 属性被一个像素化的值（每一个像素都缩放为一个正方形）替换。此外还增加了 `crisp-edges` 选项，用于让边缘更加平滑并维持高对比度。

在编写本书之时，推荐使用 `optimizeQuality` 选项来平滑插入照片，在现有的浏览器中它被默认的 `auto` 值覆盖。目前正在讨论设置一个单独的 `smooth` 属性，来和 `auto` 属性加以区分。

一组不同但有关的属性集被称为颜色插值指令，我们将在第 3 章中深入讨论。颜色插值的设置不（应该）是建议而是必要的。但是，给 `color-rendering` 设置 `optimizeSpeed` 值时，如果它减缓了渲染速度，浏览器可以忽略颜色插值模式。在实践中，对颜色插值选项的支持程度较低，导致这个区别是无关紧要的。

第 3 章

创建颜色

本章我们将更加深入地研究使用纯色来填充图形时可以选用的值有哪些。首先概述颜色在 Web 上的工作原理，然后描述在 Web 上定义颜色的不同方式：从非常可读（但不是很合理）的颜色关键词，到 RGB 和 HSL 颜色函数。

我们将在第 4 章中讨论部分透明颜色，作为颜色基础知识的一个补充。颜色的概念也是从第 6 章将要开始介绍的颜色渐变的一个必要前提。

3.1　使用名称生成朦胧玫瑰红

当代码是写给他人看（例如本书中的例子）时，使用人类可读的颜色名是非常好的做法，比如 `red`、`gold` 和 `aquamarine` 等。

如果计算机也可以识别这些颜色名那就再好不过了。值得高兴的是，在 SVG 中是可以做到的。Web 浏览器和 SVG 编辑器都可以理解 `red`、`gold` 和 `aquamarine` 的含义。它们甚至还能识别更加稀奇古怪的名字，比如 `mistyRose`、`peachPuff` 和 `mediumSeaGreen`。

这些名字是怎么来的呢？它们有两个来源：在早期的 HTML 和 CSS 版本中引入的简单的颜色关键词集，以及从 Unix 电脑 X11 窗口系统中的 SVG（以及后来的 CSS）开始使用的更广泛的关键词集。[1] 两种关键词集在所有

注 1：如果你对这些关键词如何最终成为 X11 首要标准感兴趣，Alex Sexton 从以前的 Unix 论坛中挖掘了相关历史。你可以在线观看他在 2014 年 CSSConf 上的演讲“Peachpuffs and Lemon Chiffons”（https://www.youtube.com/watch?v=HmStJQzclHc）。

主流的 SVG 查看器中都支持。此外，它们还可以在所有还在使用的浏览器（最古老的除外）的其他 CSS 属性中使用。

它们对于用户是友好的，但关键词系统也有许多限制。

首先，这 147 个关键词只能描述现代电脑显示器可以显示的数百万种颜色中的一小部分。

这些关键词的选择也有一些不一致和随意的情况。最初的网页中的颜色和 X11 系统中的颜色有时是冲突的。颜色 cyan（X11 中）和 aqua（CSS 1 中）是相同的，但和 aquamarine 不同。颜色 darkGray（X11 中）实际上比 gray（CSS 1 中）还浅。

> 所有的 gray 关键词同样也可以拼写为 grey。这是一个特性，而不是 bug。但是，一些老的浏览器只支持美式拼写方法 gray，所以建议使用它来获得更好的支持。

X11 颜色名称本身不是很系统，一些使用 dark、medium 和 light 来当前缀，而其他的还会有 pale 前缀。这些变形不总是符合逻辑的，比如 darkSeaGreen 不是真的比 seaGreen、lightSeaGreen 和 mediumSeaGreen 颜色深，而仅仅是暗一点。

不过，如果你很享受使用可读颜色名称带来的便利，所有可以识别的关键词已经在附录 A 中列出，同时还列出了对应的色值。图 3-1 显示了颜色按照字母顺序从 AliceBule 到 yellowGreen 拼起来的图片。

关键词的名称和大多数 CSS 关键词一样，是不区分大小写的。如果你觉得你的颜色非常强，可以把它们全部大写。大部分官方参考文档都使用小写，但在本书中采用驼峰命名法（关键词首字母小写，后续每个单词仅首字母大写）以使其更易于阅读。

例 3-1 是创建图 3-1 效果的代码。它使用 XMLHttpRequest 来加载一个单独的文件，文件内容是按照字母顺序排序的颜色关键词列表，然后创建长方形并分别用这些颜色填充。每个长方形都有一个子元素 <title> 来包含颜色名称，如果你在浏览器中运行代码，当鼠标在色块上悬停时，它会像工具一样给你提示。

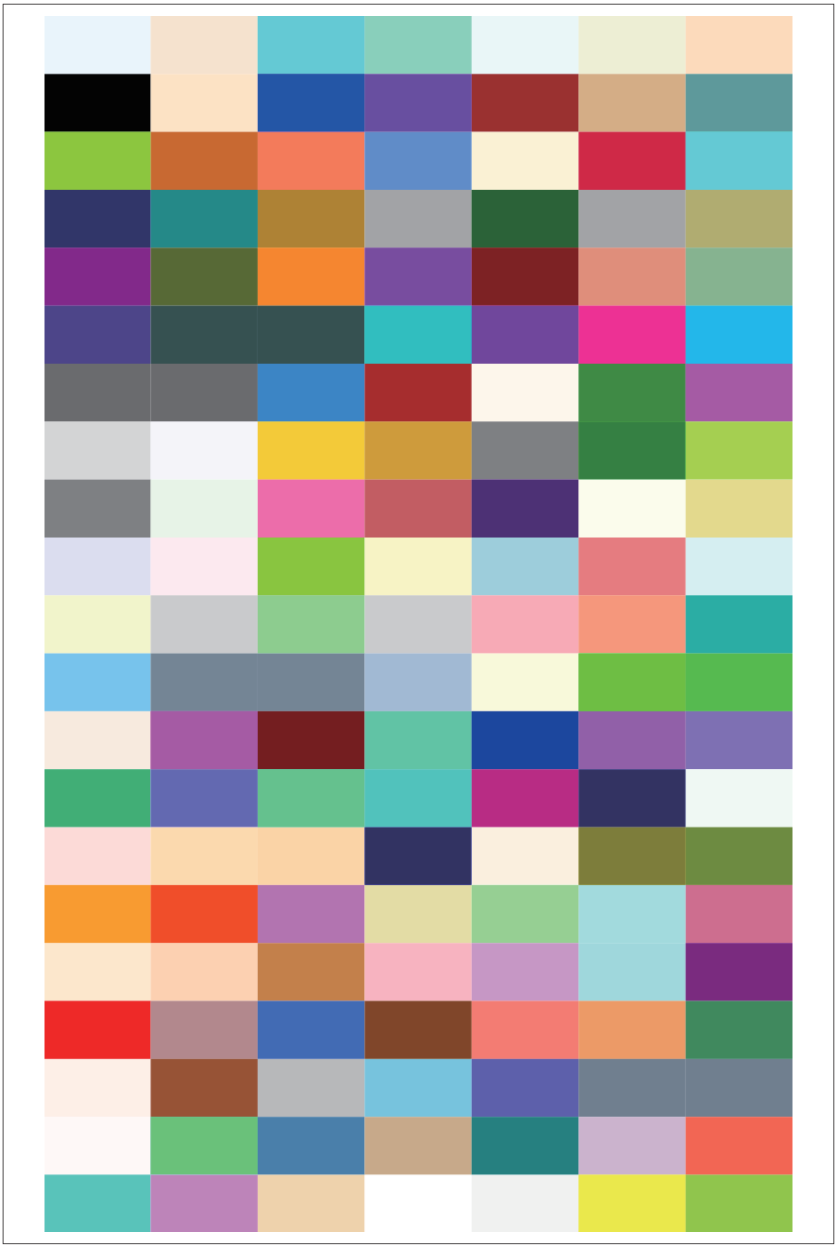


图 3-1：所有可以用颜色关键词命名的颜色

例 3-1　创建一个颜色关键词拼图

SVG 标记：

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="400px" height="650px"
     xml:lang="en"
     viewBox="0 0 7 21" preserveAspectRatio="none" >    ❶
    <title>SVG Color Keywords</title>

    <script><![CDATA[
        /* script goes here */                           ❷
]]> </script>
</svg>
```

❶ 在 SVG 规范中定义了 147 个关键字（包括不同拼写的 gray），恰好可以分布在 7×21 的网格中。viewBox 创建了一个 7 列 21 行的网格，并且使用 preserveAspectRadio="none" 来使网格拉伸到填充整个 <svg> 元素。

❷ 整个图形是通过脚本绘制的。由于这是一个 SVG 文件，所以需要使用 XML 的 <!CDATA[...]]> 块来包含脚本中的一些特殊字符。

JavaScript 代码：

```
(function(){
    var svgNS = "http://www.w3.org/2000/svg";
    var xlinkNS = "http://www.w3.org/1999/xlink";
    var svg = document.documentElement;

    var dataFileURL = "color-names.csv"                      ❶
    var request = new XMLHttpRequest();
    request.addEventListener("load", draw);
    request.overrideMimeType("text/csv");
    request.open("GET", dataFileURL);
    request.send();

    function draw() {                                        ❷
        var w = 7; //swatches per row

        var colors = request.responseText.split("\n");      ❸

        for (var i=0, n=colors.length; i<n; i++){
            var c = colors[i].trim();                        ❹

            var swatch = document.createElementNS(svgNS, "rect");
            swatch.setAttribute("width", 1);
            swatch.setAttribute("height", 1);
```

```
            swatch.setAttribute("x", i % w );
            swatch.setAttribute("y", Math.floor(i / w) ); ❺

            swatch.style.setProperty("fill", c);         ❻

            var tip = document.createElementNS(svgNS, "title");
            tip.textContent = c;
            swatch.insertBefore(tip, null);              ❼

            svg.insertBefore(swatch, null);              ❽
        }
    }
})();
```

❶ XMLHttpRequest 对象用于加载包含颜色名称列表的文件。事件监听器用于在文件加载完成时调用我们的绘画函数。MIME 类型 "text/csv" 表明我们期望的是一个分隔的文本文件，这样浏览器就不会尝试把它当作 XML 文件来解析。

❷ draw() 函数在请求的文件下载完成时调用。数据文件中每一个关键词都是单独的一行。它是通过在电子表格中插入一列，然后保存为字符分隔值（comma-separated values，CSV）文件创建的。

❸ 请求的 responseText 属性用于把整个文件作为一个单独的 JavaScript 字符串。split(token) 方法根据 token 把字符串拆分为一个字符串数组——本例中，换行符在 JavaScript 转为 \n。

❹ 使用 for 循环遍历数组（即数据文件中的每一行）中的每个字符串。trim() 方法用于去掉字符串两端多余的空白。

❺ 在拉伸的 SVG 坐标系中每一个长方形的宽和高都设置为 1；水平和垂直的位置是通过数组索引和每行色块的个数来计算的。

❻ 使用关键词名称来给 fill 属性设置值。

❼ 关键词还被用作 <title> 元素的文本内容，之后把它添加到每个 <rect> 元素的子元素中来创建提示工具。

❽ 最后，设置了样式的 <rect>（以及它的提示工具）被添加到 SVG 中。

从图 3-1 中我们可以清晰地看出，颜色关键词并不是所有可能颜色的一个代表。除了重复的灰色，还有许多灰白色的颜色，以及相对较少的暗色调。

显然，我们有比这 147 个关键词更多的可选值。想要完全了解自定义颜色的原理，我们首先要了解一点物理学和一点生物学。

3.2 彩虹三原色

在物理学中，颜色是表示波的频率或者光的能级的一个属性。可见光本身是由更宽的电磁波谱内的一段特定范围的频率组成的。能量较低时，电磁辐射就形成了无线电波；而能量较高时，它就变成了 X 射线。

光谱是连续的，颜色之间并没有明确的界限，刚好是一条平滑过渡的彩虹，包含从红色到橙色、黄色、绿色、蓝色、紫色，再到我们肉眼看不到的紫外线。每一种颜色都和特定频率和能级相关联。

正如你可以调节收音机的频率使它对某一无线电频率敏感一样，有色颜料对特定频率的光十分敏感。它们吸收特定频率（颜色）光的能量并反射其他的光。同样，化学反应中产生光是因为反应释放的能量达到了特定的频率。热光源还可以被调节来调整颜色。如果火焰燃烧反应的效率随不同燃料和氧气水平不断增长，光子（每单位光）的能量也随之增加，并且颜色会从暗红（像烧红的煤一样）变为黄色，最后变为明亮的蓝色（就像图 3-2 显示的煤气燃烧一样）。



图 3-2：煤气燃烧产生的蓝色火焰（公共领域的图片由维基共享资源用户 Sapp 提供）

无线电台的频率会混合，而光的频率不会。每个光子都保持着它自己的能级和颜色。你可以通过棱镜或者细雾来把太阳光的多种颜色折射为彩虹。

那么为什么平常我们看到的太阳光是白色而不是彩色的呢？这就不得不提到生物学。

你的眼睛对一定范围内的电磁波谱（我们称为可见光的部分）是敏感的，这是因为色素分子吸收光的能量并在大脑中将其转换成化学信号。大多数人眼中有四种感光色素：一种非常敏感但不是特定颜色的色素（负责夜视和一些运动检测），以及三种颜色色素。每种颜色色素对光谱中不同但重叠的区域敏感。

因此，我们的眼睛不是依据所有可能频率的连续光谱来识别颜色的，而是只看三种颜色色素分别吸收了多少光，你的大脑会把这些信息转化为你能看到的所有颜色。

我们眼中的色素通常和蓝色、绿色和红色相关，可以简单描述为：绿色和红色色素同样对蓝光敏感，三种色素都对中绿色的光敏感。另一种命名系统把它们描述为 S（短波长）、M（中等波长）以及 L（长波长），这是因为光的频率通常由光的波长来决定。

> 因为光具有固定的波速，所以频率（每秒的波数）和波长（相邻波之间的距离）可以互换。光的能量越高，它的频率越快，波长越短。

由于眼睛看到的颜色是三种色素不同值的混合，如果不从物理学角度来看，我们可以在绘画中混合颜色。图 3-3 概述了混合颜色的原理。如果你眼睛看到的是纯黄色的光，那么 M 和 L 色素同等程度地被触发，而 S 色素没有被触发。你的大脑根据这些信息在脑中构建出黄色。

如果你的眼睛在相同位置上看到的是亮绿色和亮红色的光的混合物，同样的色素就会被激活，你的大脑仍然认为看到的是黄色。

几乎每一种使用颜色来传递信息的设备，包括打印机和屏幕，都利用了颜色混合。然而，打印机和屏幕涉及的颜色会有不同。

印刷颜料，就好像我们眼中的色素一样，吸收特定频率的光。当有色的墨水或涂料吸收部分照射在它上面的光时，它会从反射到你眼睛中的颜色中消除它吸收的那部分光的频率。当与其他颜色混合时，其他颜色吸收的光也会被消除。这被称为*减色混合*，也是你在幼儿园就学到的颜色混合。黄色颜料会吸收大部分蓝紫光，反射绿色、黄色和红色；蓝色颜料将会吸收红色。混合黄色和蓝色，结果会吸收红色和蓝色，反射你所看到的绿色。

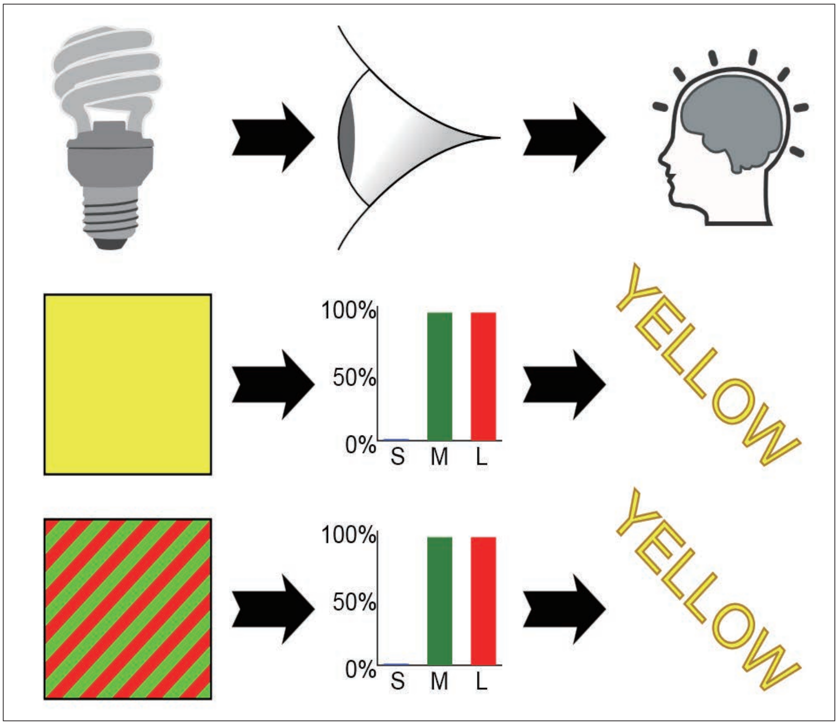


图 3-3：三色系统是如何欺骗我们的大脑的（图标由用户 jhnri4、pnx 和 benoitpetit 剪切自 openclipart.org）

也许从幼儿园开始你就记得混合深色的颜料会造成一片混乱。所以现代打印机会使用明亮的颜料，它反射的光会多于吸收的光：青色（蓝绿色）、品红（反射蓝色和红色）和黄色（反射红色和绿色）。混合这三种颜色，所有颜色的光都会被吸收，这就形成了墨黑色；其实打印机通常会包含一个单独的浓黑色。打印高品质的图形会使用一种 CMYK（Cyan、Magenta、Yellow 以及 blacK）颜色模型，它会使用四种墨来生成每种颜色。

相比之下，电脑屏幕会直接把光照射到我们的眼中。多种颜色组合来增加进入你眼中的光的总量。因此它被称为*加色*模型。彩色显示器（就像更早的彩色电视一样）使用红色、绿色和蓝色频率的光来使你眼中色素的差异最大化。精心排列的红色、绿色和蓝色的光组成的每个像素都被变换来在你眼中重现几乎所有鲜活的图案，就像自然光那样。

由于你眼中的不同色素对重叠图案的颜色敏感度不同，有一些颜色（某种饱和度的绿色，特别强的蓝色和深红色）无法使用 RGB 光来准确地表达。同样，彩色打印方式永远不能完全重现某些自然颜色。可以被一种颜色系统表示的所有可能颜色的范围被称为该颜色系统的色域。

数字图形系统中一个像素里的每种颜色的亮度水平都有一个固定的数值。早期的彩色电脑中，每种颜色有四个等级，一共包括 64（4×4×4）种颜色。HTML 中最早使用的“网络安全”颜色关键词列表可以被映射到 64 色显示器中。然而，大多数现代电脑可以支持每种颜色 256 个级别（0~255），超过 1600 万种组合。这是在 Web 中使用颜色编码的基础。

3.3　自定义颜色

在 CSS 和 SVG 中使用一组 RGB 值来自定义颜色有两种方式：

- 函数表示法，格式为 `rgb(red,green,blue)`
- 十六进制表示法，格式为 `#RRGGBB` 或 `#RGB`

在函数表示法中使用的值可以是 0 到 255 的整数或者百分比。但不可以混合使用整数和百分比，所有的值必须使用同一种类型。

六位的十六进制格式的值同样使用 0~255 的数字，但是必须要转换成十六进制的数值。十六进制中，每一位数字可以表示 0 到 15 的数值，而不仅仅是 0 到 9（hexa 是 6，deci 是 10，所以十六进制是基于 16 的数字系统）。额外的数字通过字母 `A`（10）到 `F`（15）来表示。

三位的十六进制格式是颜色中每个值的两个十六进制数字相同时的简写。

十六进制在 CSS 和 SVG 中是不区分大小写的，`#ACE` 和 `#ace` 是相同的（代表淡蓝色，也可以写成 `#AACCEE`）。

如下的颜色定义指定了相同的 RGB 值：

- `rgb(102,51,153)`
- `rbg(40%,20%,60%)`
- `#663399`
- `#639`

> 在一些最新的浏览器中，你还可以使用关键词 `RebeccaPurple` 来表示颜色值 `#639`。将该名称添加于 CSS Color Module Level 4 中，是为了纪念非常喜欢紫色的 Rebecca Meyer，她去世于六岁生日那天。她的父亲 Eric Meyer 是 W3C CSS 工作组前成员，并出版了很多有关 CSS 的图书。它是 SVG 1 规范发布后添加的唯一一个颜色关键词。

CSS Color Module Level 3 中介绍了另一种描述颜色的方式，它基于更加常见的颜色理论而不是 RGB 电脑显示器。色相 – 饱和度 – 亮度（hue-saturation-lightness，HSL）模型把颜色描述为“纯”色和黑色、白色或灰色的混合方式。它的三个具体值如下。

色相

　使用色轮中的角度定义的纯色，角度为 0 度时是纯红色，60 度时是纯黄色，120 度时是亮绿色，300 度时是品红色，360 度时又回到了纯红色。

饱和度

　混合色中纯色的强度（用于调整亮度），0% 的饱和度会有灰色的色度，100% 的饱和度是鲜亮的颜色。

亮度

　混合色中黑色或者白色的程度，0% 的亮度是纯黑色，100% 的亮度是纯白色，而 50% 的亮度是最鲜艳的颜色。

与 RGB 值不同，HSL 值通常不唯一，不同的 HSL 值可能组合出相同的颜色。例如饱和度为 0% 时，不论色相值如何，永远显示灰色。亮度为 100% 时，不论色相和饱和度值如何，永远显示白色。

> 其他两种颜色模型容易与 HSL 混淆：色相 – 饱和度 – 明度（hue-saturation-value，HSV）模型和色相 – 饱和度 – 亮度（hue-saturation-luminance，不巧，它也缩写为 HSL）模型。色相和饱和度的定义是相同的，但第三个参数的值是不可互换的。如果你想匹配其他绘图软件中定义的颜色，要确保你们使用的是相同的颜色模型。

图 3-4 使用环形色轮来展现色相角度和颜色之间的关系，不同的半径显示的是不同的亮度，每个单独的色轮有着不同的饱和度。

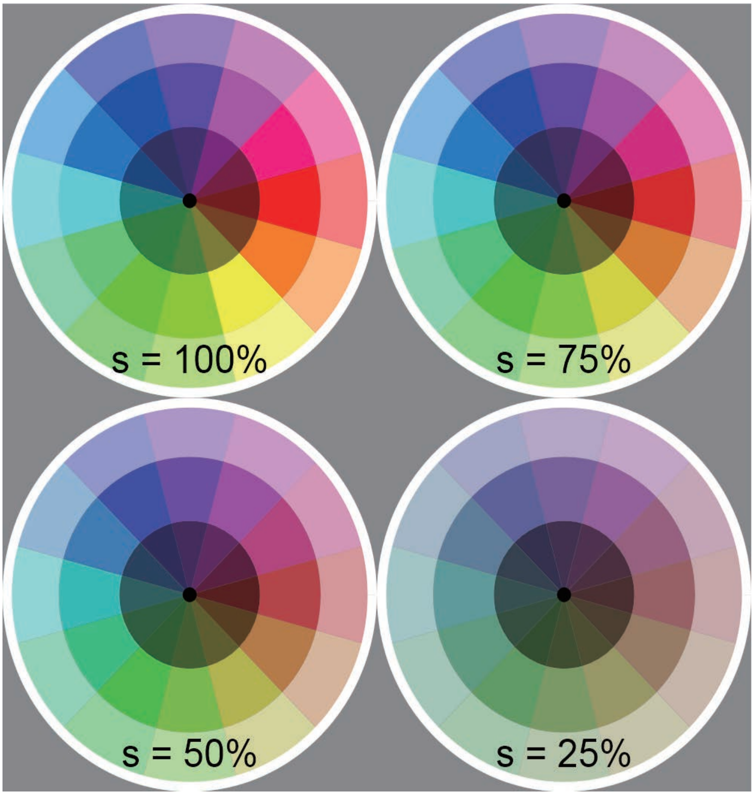


图 3-4：色相 – 饱和度 – 亮度（HSL）颜色模型；色相的值从 0 度往右按顺时针方向逐渐增加，亮度水平从内往外分别为 25%、50% 和 75%；饱和度的值如图中所示

最鲜亮的颜色是在饱和度为 100%、亮度为 50% 的时候创建的。在 RGB 模型中，这些颜色至少有一个颜色通道为 100%，一个颜色通道为 0%。通常来说，把 RGB 转换为 HSL 时：

- 饱和度的值可以计算为 100% 减去最小 RGB 通道占最大 RGB 通道的百分比

```
S = (1 - min/max) × 100%
```

- 亮度的值是最大颜色通道百分比和最小颜色通道百分比的平均值：

```
L = (min% + max%)/2
```

- 色相的值是由中间值与最大颜色通道的值减去最小通道的值的比例决定的：

```
H =   0 + 60 × ([G−B]/[R−min])，如果最大值是 R
H = 120 + 60 × ([B−R]/[G−min])，如果最大值是 G
H = 240 + 60 × ([R−G]/[B−min])，如果最大值是 B
```

该分数返回一个介于 −1 到 1 之间的值，这取决于哪种颜色的值最小，哪种颜色的值会增加或减少相对于主色通道为纯色相时的色相值。如果为红紫色，第一个公式会返回一个负的色相角度；我们可以通过给它加 360 度来转换为等效的正值。

当两个颜色通道捆绑在一起（你可以任意选择一个为最大值或最小值）的时候，转换公式仍然适用。但是当颜色为灰色时，色相的值为 undefined；颜色为黑色时，饱和度的值为 undefined。undefined 值会被设置为 0。

在 CSS 中使用 HSL 值定义颜色时，要使用 `hsl(h,s%,l%)` 函数。色相的值理论上是一个角度，但是我们在设置值的时候不需要单位。饱和度和亮度通常用百分数来表示。一些例子如下：

- 酸橙色（lime），rbg 值为 (0%, 100%, 0%)，hsl 值为 (120, 100%, 50%)
- 绿色（green），rbg 值为 (0%, 50%, 0%)，hsl 值为 (120, 100%, 25%)
- 紫色（purple），rbg 值为 (50%, 0%, 50%)，hsl 值为 (300, 100%, 25%)
- `RebeccaPurple` 色，rbg 值为 (40%, 20%, 60%)，hsl 值为 (270, 50%, 40%)

> 虽然 `hsl()` 颜色函数在所有现代浏览器中都支持（IE8 是唯一一个不支持的常用浏览器），但是一些编辑、显示或者转换 SVG 的工具可能不支持。对 CSS3 部分透明颜色的支持度与此类似，我们将在第 4 章进行讨论。

例 3-2 是创建图 3-4 的 SVG 和 JavaScript 代码。它为每个彩色的扇形部分创建了 `<use>` 元素，并且使用 `style` 对象来给 `hsl()` 函数填值。每一块都是把色相值作为 `rotate()` 变换函数的参数旋转形成的。

例 3-2　使用 SVG 和脚本创建一个 HSL 色轮

SVG 标记：

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
```

```
     width="400px" height="400px" viewBox="0 0 200 200" >
    <title>HSL Color Wheel</title>
    <defs>                                                          ❶
        <circle id="center" r="1.5" fill="black"/>
        <path id="inner" transform="rotate(-15)"
              d="M0,0L15,0A15,15 0 0 1 12.99,7.5 L0,0Z" />        ❷
        <path id="middle" transform="rotate(-15)"
              d="M15,0L28,0A28,28 0 0 1 24.25,14
                 L12.99,7.5 A15,15 0 0 0 15,0Z" />
        <path id="outer" transform="rotate(-15)"
              d="M28,0L40,0A40,40 0 0 1 34.64,20
                 L24.25,14 A28,28 0 0 0 28,0Z" />
        <circle id="edge" r="39" fill="none"
                stroke="white" stroke-width="2"/>
    </defs>
    <style type="text/css">
        text {
            text-anchor: middle;
            font-size: 8px;
            font-family: sans-serif;
        }
    </style>
    <rect fill="#888" width="100%" height="100%"/>                 ❸
    <svg class="wheel" width="100" height="100" x="0" y="0"
         viewBox="-40,-40 80,80">                                  ❹
        <use xlink:href="#center"/>
        <use xlink:href="#edge"/>                                  ❺
    </svg>
    <svg class="wheel" width="100" height="100" x="100" y="0"
         viewBox="-40,-40 80,80">
        <use xlink:href="#center"/>
        <use xlink:href="#edge"/>
    </svg>
    <svg class="wheel" width="100" height="100" x="100" y="0"
         viewBox="-40,-40 80,80">
        <use xlink:href="#center"/>
        <use xlink:href="#edge"/>
    </svg>
    <svg class="wheel" width="100" height="100" x="0" y="100"
         viewBox="-40,-40 80,80">
        <use xlink:href="#center"/>
        <use xlink:href="#edge"/>
    </svg>
    <script><![CDATA[
        /* script goes here */
]]> </script>
</svg>
```

❶ 所有的形状都预定义以方便多次使用。它们被居中绘制在一个 80 × 80 的坐标系中，每一个完整的轮子是一个半径为 40 的圆。

❷ 楔形部分使用 path 标记以及一点三角形学来创建。每一个楔形是一个 30 度的部分，以 *x* 轴为中心。为了减少对三角形学的依赖，定义楔形时首先让它的一条边与 *x* 轴平行，然后逆时针旋转，直到它的中心落在 *x* 轴上。这样，每个圆弧的一个终点的 (x,y) 坐标可以定义为 $(r,0)$，其中 *r* 为圆的半径。另一个终点的坐标可以表示为 $(r \times \cos(30^\circ), r \times \sin(30^\circ))$，它相当于 $(r \times 0.866, r \times 0.5)$。

❸ 使用以 50% 的灰色（`#888`）填充的长方形给图片提供背景。

❹ 每一个色轮都使用一个 `<svg>` 元素来定义，它的位置在图片内部，并且使用 `viewBox` 属性来确保它以坐标系的原点为中心、大小为 80 × 80。

❺ 每一个 `<svg>` 都包含一个黑色的中心点以及轮子白色的边缘，它们都不需要任何特殊的计算。

JavaScript：

```
(function(){
    var svgNS = "http://www.w3.org/2000/svg";
    var xlinkNS = "http://www.w3.org/1999/xlink";

    var wedge = 30; //angle span of each pie piece, in degrees
    var saturation = ["100%", "75%", "50%", "25%"];                  ❶
    var lightness = {outer:"75%", middle:"50%", inner:"25%"};        ❷

    var wheels = document.getElementsByClassName("wheel");           ❸
    var h,s,l,w,p,u;
    for (var i=0, n=wheels.length; i<n; i++){
        w = wheels[i];
        s = saturation[i];                                           ❹
        for (h=0; h < 360; h += wedge ) {
            for (p in lightness){                                    ❺
                l = lightness[p];
                u = document.createElementNS(svgNS, "use" );
                u.setAttributeNS(xlinkNS, "href", "#"+p );
                u.setAttribute("transform", "rotate("+h+")" );       ❻
                u.style.setProperty("fill", "hsl("+[h,s,l]+")" );    ❼
                w.insertBefore(u, w.firstChild);                     ❽
            }
        }
        var t = document.createElementNS(svgNS, "text" );            ❾
        t.textContent = "s = "+s;
        t.setAttribute("y", "35");
        w.insertBefore(t, null);
    }
})();
```

❶ 每个色轮对应的饱和度的值存储在一个数组中，随时准备好分配给每个单独的 `<svg>` 元素。由于我们不需要对这些值进行数学计算，所以可以把它们存为带有 `%` 的字符串。

❷ 我们使用存储键值对的对象来保存不同楔形部分的 `id` 以及对应的亮度的值，而不是使用两个单独的数组。

❸ 通过类名选中嵌套的 `<svg>` 元素。

❹ 在 `for` 循环的每个周期内通过索引来获取每个 `<svg>` 以及对应的饱和度值。然后嵌套使用一个 `for` 循环来循环色相的值，每一次循环给每个楔形增加 30 度。

❺ 最后，另一种类型的 `for` 循环用于遍历 `lightness` 对象中的数据键。`p` 变量在每次循环中被设置为键字符串（例如，`"outer"`、`"middle"` 和 `"inner"`）。键可以用于从对象中获取对应的值，也可以直接设为 `xlink:href` 属性的值。

❻ 色相值用于旋转楔形。由于色轮使用的是中心坐标系，所以可以简单地绕着轮子的中心旋转。`transform` 属性的值是转换函数 `rotate(h)` 组成的一个字符串，圆括号是字符串的一部分，而不是 JavaScript 的函数的一部分。

❼ 内联样式属性 `fill` 通过一个包含 CSS `hsl()` 函数的 JavaScript 字符串来创建，同样它也包括圆括号。通过把一组值放到一个数组来创建逗号分隔的参数列表；当一个数组与字符串连接时，最终会等效地输出数组中的每个值并用逗号隔开。

❽ 之后，楔块被插入为 `<svg>` 的第一个子元素，所以它会被绘制到黑色中心点以及白色边缘之下。

❾ 最后的代码块在每个 `<svg>` 中都只会运行一次，并创建一个文本标签。它们被插入到所有其他元素的后面（nothing 或 `null` 之前），所以文本会显示在色轮的最上面。

引入 HSL 颜色函数是为了更易于创建配套的颜色：色相值相同但亮度或饱和度值不同，或亮度和饱和度值相同但色相值不同。选择 HSL 模型的另一个原因是浏览器可以在 HSL 和 RGB 值之间快速转换，当然它也有自己的局限性。色相色轮在作为子元素混合渲染的时候，与我们知道的红 – 黄 – 蓝色轮不太相符。更重要的是，亮度模型不能反映出同样强烈但色相值不同的颜色之间明亮程度的区别。

以上原因以及计算机硬件的历史导致一些优化成为了所有图形处理器的标准，渐变和混色使用的是更加复杂的颜色模型。

3.4　混合和搭配

眼睛对光的敏感度并不直接和电脑显示器上像素的亮度相对应。相比于白色与浅灰色之间的变化，你更容易察觉黑色和深灰色之间的变化。对于颜色，完全饱和的蓝光看起来比相同强度的红光更暗，并且两者都不像全亮度的绿光那样明亮（从大脑反应的角度来说）。

因此，混合等量的红色和蓝色（或蓝色和绿色）的光不会创建出我们大脑认为的两者中间的一个颜色。这同样适用于几乎所有其他颜色的组合，以及灰度梯度。

电脑显示器，尤其是 Web 刚刚流行时人们所使用的 CRT 显示器，会使每一种颜色值的真实亮度失真。标准不统一导致同样的颜色在不同的显示器上看起来有很大不同。

当在计算机图形中混合颜色时，大多数现代电脑显示器使用的是 sRGB（有时被称为标准 RGB）模型。它定义了把 RGB 显示的颜色转换为标准的、在 sRGB 兼容的电脑显示器上可以察觉到亮度差异的方法。

sRGB 的使用主要影响颜色的混合。图 3-5 对比了使用 sRGB 模型来混合颜色或灰度梯度的结果与基于简单的线性算法来计算中间颜色（每一对中下方的渐变）的结果。

sRGB 模型针对用于显示图片的设备类型及其亮度做了很多假设。另外，人与人在颜色感知上不尽相同，更不用说存在色盲（眼中缺少一种或多种感光色素）等因素。也许你并不认为 sRGB 模型比线性模型好，更别说认为它是最好的混合方式。但是它已经成为计算机图形的标准，尤其是在 Web 上。

sRGB 模型以及一般的 RGB 色彩空间在图像打印领域并不受欢迎，因为它们不能编码高质量打印机所能创建的所有颜色。SVG 规范声明了为确保高保真的颜色印刷结果而使用其他颜色定义的方式。这些颜色定义使用国际色彩联盟（International Color Consortium，ICC）定义的颜色配置文件。ICC 由各大数字图像公司联合创建，它的 SVG 颜色配置在 Web 浏览器中并没有实现。



图 3-5：用 sRGB 色彩空间混合颜色（每一对中上方的渐变）与线性 RGB 混合（每一对中下方的渐变）的对比，包括针对完全饱和的颜色以及灰度的比较

CSS 以及大部分 SVG 默认使用的都是 sRGB 模型。SVG 规范定义了 `color-interpolation` 属性，它允许使用者切换到更简单的数学线性混合颜色模式（`linearRGB` 模式）。当图形由许多高亮度、高饱和度、色相差别很明显的颜色组成时，这可能是更好的选择。sRGB 模型在混合差异明显的明亮颜色时，会在生成一种介于两种颜色之间的令人不快的暗色，如图 3-5 中第一部分所示。

> 编写本书时，所有主流 Web 浏览器都不支持 `linearRGB` 颜色混合。图 3-5 是使用 Apache Batik SVG 查看器生成的。即使是 Batik 也仅仅支持线性渐变混合，而不支持分层颜色以及部分透明颜色的混合。

sRGB 被广泛应用，但滤镜计算是一个例外。SVG 滤镜允许你直接操作 RGB 颜色通道，并且默认使用 `linearRGB` 来计算。将 `color-interpolation-filters` 属性值设置为 `sRGB` 会获得更好的浏览器支持，从而在滤镜中启用 sRGB。

以上两种颜色插值模式选项都是样式属性，我们可以在单个元素或整个 SVG 中使用表现属性或 CSS 规则来设置。渐变和滤镜所使用的属性来自渐变或滤镜的元素，而不是当前正在被渲染的元素。

对于不同颜色的相对亮度，某些滤镜操作和 SVG 遮罩使用的是它们自己的修正方式。这些修正都是基于颜色本身的*亮度*（luminance）。亮度用于度量明亮程度，它不同于 HSL 颜色函数所使用的*亮度*（lightness）值。相反，前者是基于这样一个事实：完全饱和的绿色和黄色在感知上比完全饱和的红色更加明亮，而完全饱和的红色比完全饱和的蓝色更加明亮。

在一些浏览器中，遮罩从颜色到亮度的转换会受到 `color-interpolation` 属性的影响，因此默认情况下它与基于亮度的滤镜不同。

这些工具所使用的亮度调整和 sRGB 调整具有相似的作用，但不完全等价。sRGB 影响灰色或具有不同亮度（lightness）值的颜色的等级，而亮度（luminance）的权重只会影响色相的不同。此外，后者给每种颜色通道都创建了一个线性的等级，而 sRGB 定义了颜色值和感知亮度之间的曲线（伽马调整）关系。

所有这些对你有什么影响呢？大多数情况下，它仅仅是对你的一个提醒。如果你在脚本中计算颜色，或者使用滤镜操作颜色，要知道简单地求 R、G、B 的算术平均值会创建一个与浏览器混合渐变或结合部分透明的颜色不同的结果。同样，使用滤镜改变色相（亮度调整，有时也可能是 sRGB 调整）与直接在 `hsl()` 颜色函数中改变色相的值也会有不同的结果。[2]

本章的内容大多都是理论知识。我们在后面讲解透明度（第 4 章）和渐变（第 6~9 章）的时候，将会看到颜色混合的效果。

注 2：感谢 Noah Blon 让我注意到 `hsl()` 颜色函数与改变色相角度的滤镜操作之间的差异。

第 4 章

透明

纯色在图形绘制中确实占有一席之地，但是对于许多图形，你可能想增加一点微妙的变化。添加透明就是其中的一种方式，它允许你渲染一个不完全遮挡下面内容的图层。

从某种意义上说，透明是颜色的另一面，也是一个独特且更加复杂的话题。这体现在定义透明度的不同方式上。本章将对比这些方式。

当我们谈到透明度和网页设计时，有一点是不变的：我们不探讨*透明度*，而是讨论*不透明度*。这两个概念是完全相反的：当一种事物完全不透明时，它的透明度为 0；当一种事物的不透明度为 0 时，它完全透明且不可见。

4.1　穿透样式

SVG 使用三个不同的属性来控制基础形状和文本的不透明度：`opacity`、`fill-opacity` 以及 `stroke-opacity`。我们可以通过表现属性或 CSS 样式规则来给它们设置值。

CSS Color Module Level 3 扩展了 `opacity` 属性，使它可以应用到所有内容上。此外，它没有为 CSS 渲染的每一面引入 `*-opacity` 属性，而是引入了新的颜色函数。部分透明颜色可以使用 `rgba()` 和 `hsla()` 颜色函数来定义，它们可以用在 CSS 中所有要使用颜色值的地方。

Web 中的不透明度通常使用一个介于 0.0（不可见）和 1.0（纯色，不透明）

之间的小数来表示。当不透明度是颜色或图像的固有组成部分时，这些数字也被称为 alpha 值。`rgba()` 和 `hsla()` 中的 `a` 指的都是 alpha 通道。这两个函数都接收四个参数，而不是通常使用的三个，第四个参数是 0 到 1 之间的 alpha 值。

> 以前在 CSS 背景中有特殊含义的关键词 `transparent`，被重新定义为一个命名颜色。它的值等于 `rbga(0,0,0,0)`，它可以像其他颜色关键词一样，在支持 CSS3 颜色的浏览器中使用。

修改图形 alpha 值的最终效果取决于你使用的方法。具体说来，整体 `opacity` 属性与其他属性的工作方式明显不同。

`opacity` 属性应用在设置它的元素上——即使是 `<g>` 组、`<svg>` 或者 `<use>` 元素——并且它不会被继承。最终绘制的效果同样会作用在元素所有的子内容上，这使得整个元素更加均匀透明。

> `opacity` 的值在确定两个重合形状或同一形状填充和描边的重叠部分中每一点的最终颜色之后添加。

设置 `opacity` 的值小于 1 会创建一个层叠上下文，它所有的子内容都会被压入栈中。在 CSS 布局中，这可能会明显影响元素的位置。SVG 1.1 中没有类似的效果，但在 SVG 2 中它将会影响 `z-index` 形成的栈，并把所有的 3D 变换元素压入栈中。

与之不同的是，当你设置 `stroke-opacity` 或 `fill-opacity`，再使用 `rgba` 或 `hsla` 颜色函数时，透明效果在每个形状绘制时只作用在有颜色的部分。`stroke-opacity` 和 `fill-opacity` 两个属性默认情况下都可以被继承。

图 4-1 展示了一个使用粗的绿色描边和黄色填充来描绘的“8”与一个蓝紫色、透明度不变的椭圆形部分重叠后的不同效果。黄绿色形状分别为完全不透明（左上），以及通过 `stroke-opacity` 和 `fill-opacity` 属性（右上）、在 `<use>` 元素上添加 `opacity` 属性（左下）、用 `rgba` 颜色值绘制描边和填充（右下）三种方式设置半透明效果。例 4-1 给出了代码。

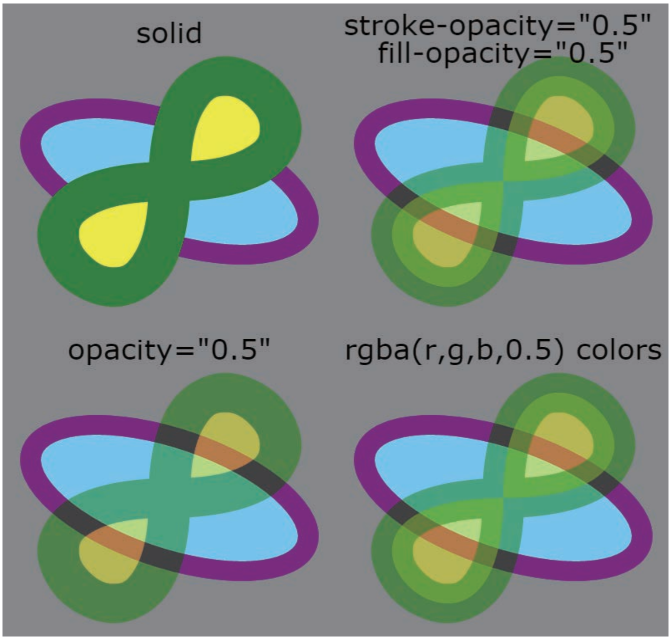


图 4-1：一个黄绿色的形状在完全不透明和三种设置部分透明的方式下的效果

例 4-1　使用不同的不透明方式来控制图形的透明度

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="400px" height="400px" viewBox="0 0 200 200"
    xml:lang="en">
   <title>Opacity Adjustments</title>
   <defs>
       <ellipse id="background"
                ry="15" rx="35" transform="rotate(20)"
                stroke="purple" fill="lightSkyBlue"
                stroke-width="5" />                        ❶
        <path id="foreground" stroke-width="10"
              d="M0,0C0,-60 60,0 0,0 S 0,60 0,0Z" />       ❷
    </defs>
    <style type="text/css">
        svg svg {
            overflow: visible;
        }
        text {
            text-anchor: middle;
```

```
            font-size: 7px;
            font-family: sans-serif;                    ❸
        }
    </style>
    <rect fill="#888" width="100%" height="100%"/>
    <svg width="100" height="100" x="0" y="0"
         viewBox="-40,-45 80,80">                       ❹
        <use xlink:href="#background"/>
        <use xlink:href="#foreground"
             fill="yellow" stroke="green" />            ❺
        <text y="-35">solid</text>
    </svg>
    <svg width="100" height="100" x="100" y="0"
         viewBox="-40,-45 80,80">
        <use xlink:href="#background"/>
        <use xlink:href="#foreground"
             fill="yellow" stroke="green"
             fill-opacity="0.5" stroke-opacity="0.5" />  ❻
        <text y="-35" dy="-0.3em"> stroke-opacity="0.5"
            <tspan x="0" dy="1em">fill-opacity="0.5"</tspan>
        </text>
    </svg>
    <svg width="100" height="100" x="0" y="100"
         viewBox="-40,-45 80,80">
        <use xlink:href="#background"/>
        <use xlink:href="#foreground"
             fill="yellow" stroke="green" opacity="0.5" /> ❼
        <text y="-35">opacity="0.5"</text>
    </svg>
    <svg width="100" height="100" x="100" y="100"
         viewBox="-40,-45 80,80">
        <use xlink:href="#background"/>
        <use xlink:href="#foreground"
             fill="rgba(100%, 100%, 0%, 0.5)"
             stroke="rgba(0%, 50%, 0%, 0.5)" />
        <text y="-35">rgba(r,g,b,0.5) colors</text>
    </svg>
</svg>
```

❶ <defs> 部分预先定义了重复使用的形状。作为背景的椭圆上，所有的渲染属性都是通过表现属性来设置的。

❷ 前景 <path> 指定了 stroke-width，它的绘制样式将会通过其他方式继承。

❸ CSS <style> 块用于设置文本标签的样式以及防止嵌套的 <svg> 元素被裁剪。注意 7px 的 font-size 将在局部坐标系中解析，所以不会产生异常小的情况。

❹ 每个嵌套的 <svg> 在主要图像的不同象限中重复创建了相同的局部坐标系，所以复用的图形可以使用相同的方式定位。

❺ 背景元素在每个样本中都复用一次，并且把前景元素层叠在它的上面。在第一个例子中，给前景元素设置了纯色的填充和描边。

❻ 接下来的 <svg> 元素定位在图像的其他象限中，并且给前景 <use> 元素设置不同的表现属性。

❼ 对于 opacity 的设置，它们将作为一个结合的组直接作用在 <use> 元素上，而不是继承到复用图形上。

图 4-1 还展示了我们在第 2 章讨论的一些概念。描边以每个形状的边缘为中心。当描边为部分透明时（由于使用 stroke-opacity 或 rgba 颜色函数），描边的内半部分的填充就会可见，这会创建一个双色效果。

另一个需要注意的特点是，描边的重叠部分并没有添加不同的色调，描边区域只会绘制一次。从概念上讲，我们往往认为描边是使用马克笔或笔刷沿着线的外部绘制的。如果确实是这样的话，那些描边两次的区域将会有两倍强度的颜色，就像你使用半透明的水彩颜料，在相同的区域刷两次那样。然而，计算机是先计算出描边总区域，然后再均匀着色，就像它是从一块半透明的塑料上裁剪下来的一样。

控制不透明度的方式有多种，那么把它们结合起来使用会产生什么样的效果呢？最终效果会叠加。例如，下面的圆通过两种不同的方式添加部分透明的填充效果：

```
<circle r="10" fill="hsla(240,100%,75%, 0.5)"
               fill-opacity="0.6">
```

它的效果与以下代码完全相同，因为 0.5 × 0.6=0.3：

```
<circle r="10" fill="hsl(240,100%,75%)"
               fill-opacity="0.3">
```

它得到的不透明度更加复杂，因为透明度是在合并其他颜色之后应用的。尽管如此，在其他属性导致的不透明度级别改变之后，0.5 的不透明度属性值依然会导致形状中每个像素的 alpha 值减半（乘以 0.5）。

由于 opacity 属性只是对每个像素的值进行简单的数学上的调整，使用大多数显卡的 GPU 来实现会很高效。不透明度的改变往往会使动画更加平滑，尽管不是所有的浏览器都为 SVG 内容采用这一点。

部分透明颜色 fill-opacity 和 stroke-opacity 不会进行同样的优化，因为它们都仅仅影响元素的一部分。但是，如果你不是动态改变不透明度的值，这些属性可能会更高效，因为它们不会强制创建层叠上下文。

4.2 其他效果

正如 3.4 节中所提及的那样，部分不透明对象创建的颜色是基于 sRGB 模型计算的。根据该规范，它应该会受到 color-interpolation 模式的影响，但大多数查看 SVG 的软件都默认使用 sRGB 并且仅仅支持这一种模式。

多层重叠显示的最终颜色需要经过以下几个步骤：首先依据 sRGB 模型对两种颜色进行缩放，然后获取背景和前景（alpha 值是前景的一个权重）的加权平均值，最后反转 sRGB 缩放。该计算不会在意背景颜色是单一元素创建的还是与部分透明元素混合创建的。

这种混合颜色的方法被称为简单 alpha 合成。在许多图形程序中，它也被称为“normal”混合模式。

聚焦未来

简单 alpha 混合之外

许多图形程序定义了另一种控制两个不同层或对象的颜色如何结合在一起的混合模式。

在 SVG 1.1 中，有多种混合模式可供选择，但是只能在滤镜中使用。W3C 创建的新 Compositing and Blending Specification 中将 CSS 以及 HTML5 canvas 使用的混合模式的定义标准化。它还介绍了两个可以应用到 SVG 上的新 CSS 属性：mix-blend-mode 和 isolation。

元素上的 mix-blend-mode 属性用来控制元素如何与它后面的颜色混合。一共有 16 种不同的模式（包括默认的 normal），分别定义了组合元素的 RGBA 或 HSLA 值与混合背景内容对应的值的计算方式。

组元素或 <use> 元素上的 isolation 属性用于限制混合作用的范围。隔离元素在子内容互相混合之后，再使用自己的混合模式来把子内容的混合结果与背景相融合。你可以使用 isolation:isolate 来显式地隔离一个元素，但有一些其他的 CSS 属性也可能导致隔离，比如设置 opacity 的值大于 1，使用滤镜或遮罩，在组本身上使用非 normal 的混合模式，或者使用 3D 转换。HTML 中内联的 <svg> 元素默认会被隔离。

编写本书时，混合模式在最新的 Firefox、Chrome、Opera 和 Safari 中得到了支持。但 IE 中和许多移动浏览器还不支持，微软的 Edge 浏览器正考虑加以支持。

使用简单的 alpha 混合创建的颜色直接对应于背景色和纯色对象之间进行 sRGB 渐变产生的颜色。alpha 值决定了沿渐变方向偏移的距离。如果 alpha 的值为 0，则该对象完全透明，渐变的初始值是背景色。如果 alpha 的值为 0.5，使用的是渐变的中点的颜色。如果 alpha 值为 1（对象完全不透明），显示的颜色与渐变的终点颜色（前景对象本身的颜色）相同。

图 4-2 中，上方是红色（`#F00`）到蓝色（`#00F`）的光滑渐变，下方是在纯红色的长方形之上放置一个蓝色正方形，且逐次改变正方形的 `fill-opacity` 的值。图 4-2 展示了两者的对比效果。

你可能觉得似乎图 4-2 中的正方形也是使用渐变填充，且渐变方向相反。这是一种公认的光学错觉。你的眼睛和大脑会加强每条边上颜色之间的对比。当一个纯色元素在它的每一侧都有不同的对照颜色（或一侧有渐变），对比度的增强会让我们在感知上以为是渐变。

为了证明你看到的确实是纯色，使用几张纸（纸的颜色都相同）覆盖与正方形相邻的部分，直到你能看到一个单独的纯色方块。

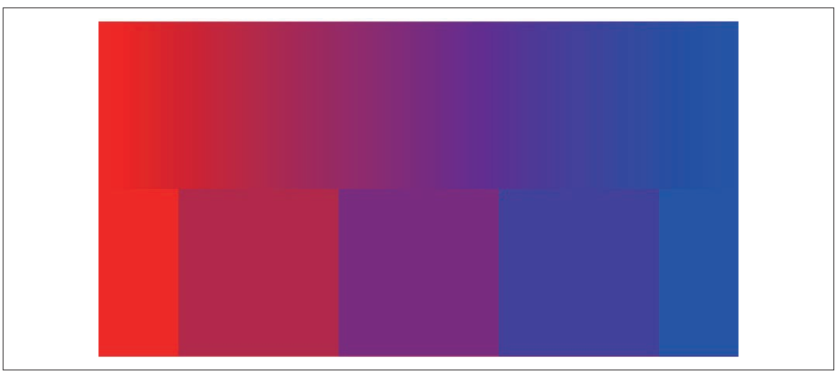


图 4-2：使用渐变和不透明度混合颜色的对比；正方形由蓝色正方形和位于其下部的纯红色长方形组成，蓝色正方形的 `fill-opacity` 值分别为 0、0.25、0.5、0.75、1

例 4-2 是生成图 4-2 的代码。

例 4-2　使用不透明度搭配渐变来创建混合颜色

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="400px" height="200px" viewBox="0 0 400 200"
     xml:lang="en">
    <title>Blending with opacity and gradients</title>
    <defs>
        <linearGradient id="gradient">
            <stop offset="0" stop-color="#F00"/>
            <stop offset="1" stop-color="#00F"/>
        </linearGradient>
        <rect id="square" width="100" height="100" y="100"/>
    </defs>

    <rect fill="url(#gradient)" width="400" height="100"/>
    <rect fill="#F00" width="400" height="100" y="100"/>
    <g fill="#00F">
        <use xlink:href="#square" x="-50" fill-opacity="0" />
        <use xlink:href="#square" x="50" fill-opacity="0.25" />
        <use xlink:href="#square" x="150" fill-opacity="0.5" />
        <use xlink:href="#square" x="250" fill-opacity="0.75" />
        <use xlink:href="#square" x="350" fill-opacity="1" />
    </g>
</svg>
```

在第 6 章中我们将会介绍图 4-2 中使用的 `<linearGradient>` 和 `<stop>` 元素，并且介绍如何利用它们创建不同的渐变效果。但首先，第 5 章中我们将探讨 SVG 中使用 URL 引用另一个元素来填充当前元素会产生的结果。

第 5 章

渲染服务

当 `fill` 或 `stroke` 的值比单一颜色更加复杂时（`transparent` 或者其他），SVG 使用一种叫作*渲染服务*的概念来描述图形是如何被渲染的。

渲染服务通过自身特定的 SVG 元素来定义。这些元素（渐变和图案）不会直接创建一个可见图形。它们仅仅用于形状或文本的 `fill` 和 `stroke` 属性中。然而，通过使用 XML 标记来定义渲染服务，它可以有无数种变化：任何 SVG 图形都可以用于生成 SVG 图案，包括其他图案本身。

相比之下，当使用 CSS 来给 HTML 内容添加样式时，所有关于如何绘制元素的信息都必须包含在 CSS 样式规则内。在 CSS 2.1 中，创建图案的唯一方式是引用外部图片文件。在此之后，CSS 已经推出许多以前只能在 SVG 中实现的图形效果，比如渐变以及改变图片位置。尽管最终的结果可能看起来很相似，但这些属性的 CSS 语法和等价的 SVG 有很多不同之处。在本书的其余部分，我们将会在“CSS 与 SVG”附注栏中对这两种方式加以比较。

本章介绍了基本的渲染服务模型，并介绍如何使用它来提供单一渲染颜色。

5.1　渲染和壁纸

所有 SVG 渲染服务的一个主要特点是它们生成的图形内容都在长方形区域内。有时这可能会有些限制，但也会有一些潜在的性能优化机会。

SVG 渲染服务不需要知道它要填充的形状是什么——它会任意地涂满整个墙面。形状本身作为遮罩或者模子来限制真实绘制的部分，正如粉刷匠在粉刷墙壁时会遮住地板、天花板、灯具等，最终被粉刷的只有墙壁。

理解渲染服务（尤其是谈论渐变和图案时）的另一种方式是把渲染内容想象成一大张墙纸。形状是从大块墙纸中裁剪出来的部分，如图 5-1 所示。

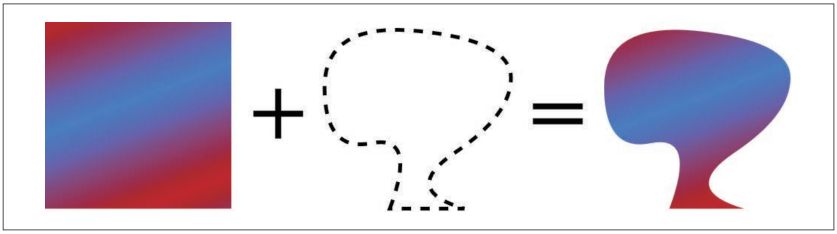


图 5-1：可以想象填充的形状是从矩形图案中裁剪出来的

当然，计算机不会使用纸和剪刀，而是把形状作为栅格形状（横向扫描）。填充区域内的每一个点时，计算机都会从渲染服务中查找对应的 (*x*, *y*) 坐标。所以，渲染服务可以是能够给每一个 (*x*, *y*) 值都分配特定颜色的任意对象。

理论上说，“渲染”可以是任何事物：一种单独的颜色、一个或多个渐变、重复的图案、位图、文本，甚至是其他 SVG 文件。实际上，SVG 1.1 有两种类型的渲染服务：渐变和重复图案。然而，这些核心元素可以创建上述所有选项，本书的其余部分将会对此进行说明。

5.2 标识资源

“服务”意味着它是多个资源的外部引用。理论上说，你可以单独创建一个文件来包含你所有的渲染服务，并且在 `fill` 和 `stroke` 属性中引用它，但是目前浏览器支持比较差。通常情况下，*渲染服务*指的是每一个渐变或图案对象可以给多个 SVG 形状提供渲染（渲染指令）。



在本书编写之时，外部渲染服务仅仅在 Firefox 和使用 Presto 渲染引擎的 pre-Blink 版本的 Opera 中得到了支持。

为了使用渲染服务，你需要使用 URL 语法来引用渲染服务元素，并把它放在 CSS 的 `url()` 函数内。由于浏览器支持程度的限制，这个 URL 几乎总是

一个内部引用，就像 url(#myReference) 这样。井号（#）表示接下来是一个指向特定元素的标记，事实上井号之前没有任何东西表示浏览器应该在当前文档中查找该元素。具体而言，它寻找的是一个 id 属性与标记片段相匹配的元素（即 <pattern id="myReference">）。

因此，引用一个 ID 为“customBlue”的渲染服务作为填充值如下所示：

```
<rect fill="url(#customBlue)" width="100" height="100"/>
```

因为 fill 是一个表现属性，所以你还可以在文档中任意位置使用一个 <style> 块来设置值：

```
rect {
    fill: url(#customBlue);
}
```

上面的规则将会设置文档中的所有矩形都用那个渲染服务，前提是该样式不会被更加具体的 CSS 规则覆盖。

外部样式表中的相对 URL 通常指向 CSS 文件的位置，而不是使用这些样式的文档的位置。这包括像 #customBlue 这样的本地 URL 片段，如果它定义在外部 CSS 文件中，则永远不会匹配任何东西。再加上对外部渲染服务的支持度比较差，这很不幸地意味着你不能有效利用外部样式表来设置渲染服务。

> 相对 URL 也会受到 xml:base 属性或 HTML<base> 元素的影响，这两者都会导致渲染服务引用失败。

理论上（或者你仅仅需要支持 Firefox），如果你有一组预先定义在名为 brand.svg 的文件中的颜色，你可以提供一个引用该资源的相对路径，然后使用标记片段来指向特定元素：

```
<rect fill="url(brand.svg#customBlue)"
      width="100" height="100"/>
```

或者你甚至可以提供一个绝对 URL 指向同一个资源，假设外部文件可以在你的 Web 域下安全访问：

```
<rect fill="url(//example.com/assets/brand.svg#customBlue)"
      width="100" height="100"/>
```

不幸的是，对该选项的支持程度很差，因为服务的概念可以被看作另一种资源库，一种存储常用颜色、渐变、图案、遮罩以及其他资源的单个文件。

现在，如果你有在多个 SVG 中共用的渲染服务，你需要直接把它们添加在每个文档中，要么在你的服务上使用一些预处理，要么使用 Ajax 技术来通过客户端 JavaScript 导入。

因为很多东西可能会对外部资源的加载产生干扰，甚至浏览器本身不支持，所以 SVG 的 `fill` 和 `stroke` 属性允许你定义一个备用的颜色值。该色值在 `url()` 引用之后设置，通过空白来分隔，如下所示：

```
rect {
    fill: url(brandColors.svg#customBlue) mediumBlue;
}
```

也可以使用表现属性和十六进制颜色语法：

```
<rect fill="url(brandColors.svg#customBlue) #0000CD"
      width="100" height="100"/>
```

如果引用的渲染服务不能被加载，该矩形会使用定义的纯蓝色来渲染。

聚焦未来
分层填充与降级方案

SVG 2 中引入了分层填充和分层描边，类似于 CSS 盒布局的分层背景图片。

使用多个背景图片时，多个渲染选项使用从上到下用逗号分隔的列表来定义。在列表的最后可以添加一个备用的颜色，用空白与前面的内容分隔开。

不同于 CSS 中的简写方式 `background`（设置 `background-image` 列表和单个 `background-color` 值），最后的颜色通常不会在其他渲染层之下绘制。

例子如下所示：

```
.typeA {
    fill: url(#pattern1), url(#gradient) mediumBlue;
}
.typeB {
    fill: url(#pattern2), url(#gradient) darkSeaGreen;
}
```

如果渲染服务可以正确加载，`typeA` 和 `typeB` 两个图形将会根据相同的渐变层上不同的图案层来区分。如果没有找到渲染服务（也许你的 Ajax 脚本没有正确执行），两个图形将会使用不同的纯色来绘制。

如果你确实想要在图案或渐变层之下绘制纯色，可以使用逗号来与前面的层级分隔：

```
.typeA {
    fill: url(#pattern), mediumBlue;
}
.typeB {
    fill: url(#pattern), darkSeaGreen;
}
```

本例中，两个图形的类使用了相同的图案（可能添加一个纹理效果），但是它们布局在不同的纯色之上。

5.3 纯色渐变

很多时候，尤其是在商业产品中使用颜色时，设计师将会给颜色取一个特定的名称。同样的颜色可能会在许多与品牌相关的图形中出现，例如公司 logo 的不同版本、标题文字、产品标签等。我们可以把每种颜色都只定义一次，并分别给它们分配一个名称，然后就可以在内容中直接使用该名称，而不用维护它们的 RGB 值组成的列表。这也使得在设计过程中改变某一颜色变得更加容易。

SVG 渲染服务非常适合去做这项工作。我们可以在多个图形的 `fill` 和 `stroke` 属性中引用它的 ID，而真实的颜色值只需要指定一次且很容易更新（可以通过这种方式引用动画，我们将在第 14 章介绍）。

最初的 SVG 规范没有明确包含纯色渲染服务，但是所有的浏览器都允许你使用单个不变的颜色组成的渐变来实现这种效果。例 5-1 展示了这一方式，它使用 `<linearGradient>` 元素给在（虚构的）ACME 公司的商标中使用的四种颜色命名。这些颜色最终被用于绘制公司 logo，如图 5-2 所示。



图 5-2：使用命名颜色的 ACME logo

例 5-1　使用单色渐变来给商标定义命名的颜色

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xml:lang="en"
     width="100mm" height="50mm">                        ❶
    <title>ACME Logo</title>
    <defs>
        <linearGradient id="AcmeRed">                    ❷
            <stop stop-color="#FF4022" />
        </linearGradient>
        <linearGradient id="AcmeMaroon">
            <stop stop-color="#80201C" />
        </linearGradient>
        <linearGradient id="AcmeGold">
            <stop stop-color="#FFFC32" />
        </linearGradient>
        <linearGradient id="AcmeWhiteGold">
            <stop stop-color="#FFFCE0" />
        </linearGradient>
        <symbol id="AcmeLogo" viewBox="0,-40 160,80" >   ❸
            <path d="M0,0 L40,-40 L40,-20 L160,-20
                   L160,20 L40,20 L40,40z"
                  fill="url(#AcmeRed)"/>                 ❹
            <path d="M16,-10 L35,-29 L35,-15 L155,-15 L155,-10 z"
                  fill="url(#AcmeGold)"/>
            <path d="M13,-7 L16,-10 L155,-10 L155,-7 z"
                  fill="url(#AcmeMaroon)"/>
            <text x="40" y="15"
                  style="font-family:Arial; font-weight:bold;
                         font-size:20pt;"
                  fill="url(#AcmeWhiteGold)">ACME</text>
        </symbol>
    </defs>
    <use xlink:href="#AcmeLogo" />                       ❺
</svg>
```

❶ SVG 没有设置 viewBox 属性，它的比例是由包含 logo 的 <symbol> 元素控制的。然而，默认的宽和高确保图片在插入到其他网页中时可以保持原始宽高比和合理的默认大小。

❷ 该公司有四种商标颜色：AcmeRed、AcmeMaroon、AcmeGold 和 AcmeWhiteGold。每种颜色都使用一个 <linearGradient> 元素和一个 <stop> 元素定义为渲染服务。

❸ logo 本身定义在 <symbol> 元素内，以便在其他图形中重复使用。viewBox 在竖轴中点建立了一个坐标系。

❹ symbol 内的每一个形状都使用一种预先定义的渲染服务来设置填充色。

❺ 在 SVG 内使用 <use> 元素绘制 logo。<use> 元素没有任何位置或大小属性，所以重复使用的 <symbol> 将会拉伸填充整个 SVG 区域。

仔细查看会发现每个渐变都由两个元素组成，<linearGradient> 和 <stop>：

```
<linearGradient id="AcmeRed">
    <stop stop-color="#FF4022" />
</linearGradient>
```

<linearGradient> 定义了渲染服务，并且设置了引用它所需要的 id 值。该渐变元素也包含定义颜色的 <stop> 元素。通常的渐变中会有多个 stop 来定义初始、最终以及中间的颜色。

颜色是通过表现属性 stop-color 来定义的。还有一个可选的表现属性 stop-opacity，类似于 fill-opacity 或 stroke-opacity，但颜色值默认是完全不透明的。

虽然例 5-1 在每个浏览器的测试中都按照预期的执行，但在 Apache Batik 中将会执行失败，因为它对语法的要求更加严格。为了使其能够正常工作，<stop> 元素需要一个 offset 属性，我们将在 6.1 节中讨论。

因为颜色是单独设定的，所以我们很容易统一改变它们的颜色或添加相同的动画。stop-color 是一个表现属性，你甚至不需要通过编辑 XML 来改变颜色，而是可以使用 CSS 规则来覆盖。

结果，你可以使用 CSS 条件规则来改变颜色。含有媒体查询的样式表可以用于给高质量打印机或灰度打印指定打印颜色。由于这些颜色用于在图形的其他部分进行引用，样式表不需要辨认使用每种颜色的所有元素，也不需要区分 fill 和 stroke 的值。

虽然 stop-color 是一个表现属性，但它默认是不可继承的。它必须明确地直接在 <stop> 元素上设置，或者使用 inherit 关键词设置。

例 5-2 给出了打印样式的部分样本。对于彩印，它使用 HSL 函数来重新定义颜色，然后它会被映射到打印设备使用的整个色域。对于黑白打印，它会给每个颜色分配一个灰色阴影，这会创造出比颜色自动转换为灰色更强的对比度。灰度版本的 logo 如图 5-3 所示。



图 5-3：把使用命名颜色的 ACME logo 转换为黑白色

例 5-2　为打印图形重新定义命名颜色

```
@media print AND (color) {
    #AcmeRed stop        { stop-color: hsl(10, 100%, 60%); }
    #AcmeMaroon stop     { stop-color: hsl( 0, 65%, 30%); }
    #AcmeGold stop       { stop-color: hsl(60, 100%, 60%); }
    #AcmeWhiteGold stop { stop-color: hsl(55, 100%, 90%); }
}
@media print AND (monochrome) {
    #AcmeRed stop        { stop-color: #555; }
    #AcmeMaroon stop     { stop-color: #222; }
    #AcmeGold stop       { stop-color: #DDD; }
    #AcmeWhiteGold stop { stop-color: #FFF; }
}
```

虽然大部分浏览器在打印网页时可以正确应用 CSS 打印样式，但当用户在彩色打印机上进行黑白打印时，浏览器并不总会应用黑白样式。

使用渲染样式来给非标准颜色命名还会使你的代码更易于他人阅读。使用有意义的 `id` 值，会使每个元素的颜色和用途对于将来接管你工作的任何开发人员来说都更加清楚明白。

聚焦未来

<solidcolor> 渲染服务

命名颜色的渲染服务有许多好处。但是用单色渐变来创建命名颜色多少有点儿问题，这肯定不是这些元素的最初用途。

所以 SVG 2 中使用 <solidcolor> 来创建单色渲染服务。它使用 solid-color 和 solid-opacity 表现属性来设置颜色的值。

使用 <solidcolor> 元素的话，例 5-1 中的四种商标颜色可以这样定义：

```
<solidcolor id="AcmeRed"         solid-color="#FF4022" />
<solidcolor id="AcmeMaroon"      solid-color="#80201C" />
<solidcolor id="AcmeGold"        solid-color="#FFFC32" />
<solidcolor id="AcmeWhiteGold"   solid-color="#FFFCE0" />
```

这不仅减少了标记的数量，还使得代码更容易让人理解。

<solidColor> 元素（注意大写的 C）在 SVG Tiny 1.2 规范中就已经引入了，所以它在一些图形程序中得到了支持，但是你需要在根 <svg> 元素上明确地设置属性 version="1.2"。（相比之下，Web 浏览器会忽略 version，并对所有的 SVG 内容使用最新的规范。）最新的 SVG 2 规范草案中改变了大写字母，使元素对 HTML 更加友好，不幸的是它打破了区分大小写的 XML 查看器的兼容性。

在本书编写之时，<solidColor> 和 <solidcolor> 都没有在任何主流浏览器的稳定版中得到支持。但正在开发的 Firefox 会对此加以支持。

第 6 章

简单的渐变

矢量图形通常被看做“线图”，拥有锐利的边缘以及连续的色块。这种干净、简洁的样式通常就可以满足你的设计。但是一种完整的图形语言应该有创建柔和边缘和变换颜色的选项。渐变是 SVG 中实现这种效果的最简单的方式。

渐变是从一种颜色或不透明状态平滑过渡到另一种颜色或不透明状态。SVG 目前支持两种类型的渐变：线性渐变和径向渐变。线性渐变内过渡阶段的每种颜色沿着平行线的方向延伸，而径向渐变内每种颜色都是一个环形。

本章介绍了 SVG 渐变的基础结构，以及可以创建的不同颜色变换。我们还会对比创建相似效果的 SVG 语法与 CSS 语法的不同。从现在开始，所有例子中的渐变都使用默认的方向以及比例，并利用它们填充简单的长方形。接下来的章节将会讲解更加灵活的 SVG 渐变。

6.1　逐步渐变

例 5-1 的单色渐变演示了创建一个渐变所需的最少标记。但我们没有使用任何创建渐变效果的特性。要想看到颜色过渡的效果，我们至少需要两个颜色结点（stop）。

双色线性渐变的基础结构如下所示：

```
<linearGradient id="red-blue">
    <stop stop-color="red" offset="0"/>
    <stop stop-color="lightSkyBlue" offset="1"/>
 </linearGradient>
```

两个 <stop> 元素定义了创建渐变所需要混合的颜色，填充长方形的结果如图 6-1 所示。每个 <stop> 都有一个新的属性 offset，它用于定义颜色在渐变中的位置。offset 的值可以使用 0.0 到 1.0 之间的数字或百分比来表示。本例中，渐变最开始是纯红色，最后是浅蓝色。

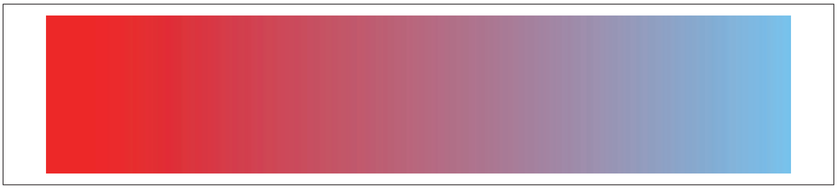


图 6-1：有两个结点颜色的渐变

当颜色结点超过两个时，offset 之间的间隔会控制颜色变化的频率。图 6-2 显示了使用不对称渐变的结果，代码如下：

```
<linearGradient id="red-blue-2">
    <stop stop-color="red" offset="0"/>
    <stop stop-color="lightSkyBlue" offset="0.3"/>
    <stop stop-color="red" offset="1"/>
</linearGradient>
```

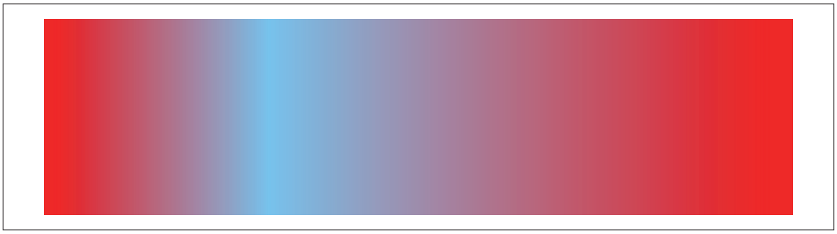


图 6-2：三个 offset 值不规则的结点创建的渐变

如果第一个 offset 的值大于 0 或最后一个 offset 的值小于 1，渐变部分的两侧将会使用纯色块填充，如图 6-3 所示，代码如下：

```
<linearGradient id="red-blue-3">
    <stop stop-color="red" offset="0.3"/>
    <stop stop-color="lightSkyBlue" offset="0.7"/>
</linearGradient>
```

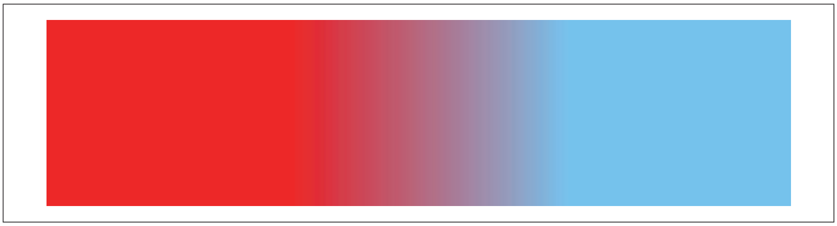


图 6-3：使用两个结点，与边缘之间有偏移的渐变

这实际上就是只使用一个 <stop> 元素所产生的效果，就像纯色渐变（例 5-1）那样：两侧的偏移使用同样的纯色填充。

> 0 到 1（或 0% 到 100%）范围之外的 offset 值将会被限制在这个间隔之内。

XML 元素的顺序很重要，<stop> 元素必须按照从渐变开始到结束的顺序设置。如果你定义的 offset 的值比之前结点元素的 offset 的值小，它会被调整为等于前面元素中最大的 offset 值。

如果连续的结点拥有相同的 offset，你将会创建锐利的边缘或条纹，如图 6-4 所示，代码如下：

```
<linearGradient id="red-blue-4">
    <stop stop-color="red" offset="0.3"/>
    <stop stop-color="lightSkyBlue" offset="0.3"/>
    <stop stop-color="lightSkyBlue" offset="0.7"/>
    <stop stop-color="red" offset="0.7"/>
</linearGradient>
```



图 6-4：使用渐变创建条纹

6.2 透明渐变

之前的例子使用的都是完全不透明的渐变颜色，但我们说过渐变也可以很好地控制不透明度。一种方法是使用 `rgba()` 和 `hsla()` 这种半透明颜色来给 `stop-color` 设值。但 SVG 有一种更简单的方式，它允许独立于颜色之外控制不透明度。

每个渐变结点的不透明度由 `stop-opacity` 属性来控制，就像所有其他不透明度属性一样，它的值是 0 到 1 之间的一个数字。同时与 `stop-color` 一样，它的值默认是不继承的。

你可以使用 `stop-opacity` 创建一种颜色从不透明过渡到透明的渐变，也可以使用以下代码实现同时变换颜色和不透明度：

```
<linearGradient id="red-blue-5">
  <stop stop-color="red"          stop-opacity="0" offset="0"/>
  <stop stop-color="lightSkyBlue" stop-opacity="1" offset="1"/>
</linearGradient>
```

图 6-5 中显示的半透明渐变是在文本“Gradients”之上填充矩形。在最左边是纯红色，但它是完全透明的。随着渐变往右进行，颜色也随着改变，同时变得更加不透明，使文本越来越模糊。



图 6-5：半透明颜色渐变

这里唯一需要的属性是 `offset`，而 `stop-color` 默认是黑色且 `stop-opacity` 默认是 1（100% 不透明，或者说没有透明度）。在许多颜色渐变中，我们不设置不透明度值而使用它的默认值，同样你也可以不设置颜色值，这样的话，渐变就会从透明黑色过渡到不透明黑色。

```
<linearGradient id="brightness">
    <stop offset="0" stop-opacity="0"/>
    <stop offset="1" />
</linearGradient>
```

该渐变如图 6-6 所示，且同样在渐变填充的矩形下面绘制文本，以此来突出

不透明度的改变。



图 6-6：单色不透明度渐变

6.3　控制颜色变换

我们可以使用更多的结点来创建更加复杂的渐变。例如，下面的例子将创建一个彩虹渐变，从品红到蓝色再到青色、绿色、黄色、红色、黑色，再回到品红，构成一个完全饱和颜色的循环：

```
<linearGradient id="rainbow">
    <stop stop-color="#F0F" offset="0"/>
    <stop stop-color="#00F" offset="0.1667"/>
    <stop stop-color="#0FF" offset="0.3333"/>
    <stop stop-color="#0F0" offset="0.5"/>
    <stop stop-color="#FF0" offset="0.6667"/>
    <stop stop-color="#F00" offset="0.8333"/>
    <stop stop-color="#F0F" offset="1"/>
</linearGradient>
```

彩虹渐变效果如图 6-7 所示。



图 6-7：使用 7 个结点形成的彩虹渐变

明确设置颜色结点是为了保证在任何时候都至少有一个颜色通道保持最大化，维持完全饱和的效果。下面的代码片段使用 HSL 函数设置颜色并使用百分比来设置偏移，定义了一个完全相同的渐变：

```
<linearGradient id="rainbow">
    <stop stop-color="hsl(300, 100%, 50%)" offset="0%"/>
    <stop stop-color="hsl(240, 100%, 50%)" offset="16.67%"/>
    <stop stop-color="hsl(180, 100%, 50%)" offset="33.33%"/>
    <stop stop-color="hsl(120, 100%, 50%)" offset="50%"/>
    <stop stop-color="hsl( 60, 100%, 50%)" offset="66.67%"/>
    <stop stop-color="hsl( 0, 100%, 50%)" offset="83.33%"/>
    <stop stop-color="hsl(-60, 100%, 50%)" offset="100%"/>
</linearGradient>
```

品红，也被称为紫红色，使用了 300 度的色相（渐变的起点）和 −60 度的色相（渐变的终点）来定义。因为他们在圆中角度相同，在色轮中对应的色相也相同。

虽然可以用 HSL 值来指定颜色，但是颜色是通过 RGB 颜色空间直线混合的，而不是通过 HSL 色轮的旋转角度来混合。正如 3.4 节所说，中间颜色受到 sRGB 模型中颜色亮度的权重影响。一旦调整为 sRGB 模型，每个通道（红、绿、蓝）的值都会单独增加或减少。

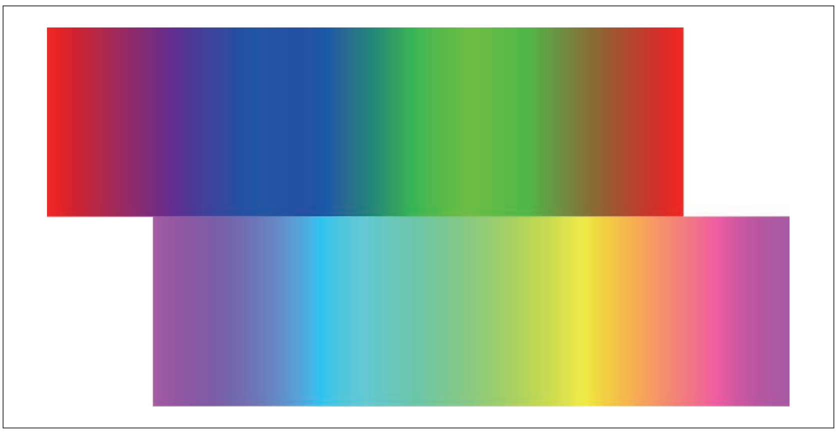


图 6-8：使用红色、蓝色、绿色结点（上）和使用品红、青色、黄色结点（下）形成的彩虹渐变

如果我们没有指定首要和次要颜色，品红（#F0F）和青色（#0FF）的中点将不是完全饱和的蓝色（#00F），而是一个淡得多的蓝色，因为它有部分红色通道和绿色通道变浅。如果你混合纯红色（#F00）和饱和的蓝色（#00F），结果将是一个深紫色（两个颜色通道都处于完全打开和关闭的中间），而不是一个完全饱和的品红。

图 6-8 比较了使用首要颜色生成的深暗色混合与使用次要颜色生成的更淡、更有生机的渐变。生成该图的完整代码如例 6-1 所示。

例 6-1　比较两个多色渐变

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="2in">
    <title xml:lang="en">Dark and Light Rainbow Gradients</title>
    <linearGradient id="primary">                                ❶
        <stop stop-color="#F00" offset="0"/>
        <stop stop-color="#00F" offset="0.3333"/>
        <stop stop-color="#0F0" offset="0.6667"/>
        <stop stop-color="#F00" offset="1"/>                     ❷
    </linearGradient>
    <linearGradient id="secondary">
        <stop stop-color="#F0F" offset="0"/>
        <stop stop-color="#0FF" offset="0.3333"/>
        <stop stop-color="#FF0" offset="0.6667"/>
        <stop stop-color="#F0F" offset="1"/>
    </linearGradient>
    <rect width="85.714%" height="1in" x="0" y="0"
          fill="url(#primary)" />
    <rect width="85.714%" height="1in" x="14.286%" y="1in"
          fill="url(#secondary)" />                              ❸
</svg>
```

❶ 虽然 <linearGradient> 元素可以包含在 <defs> 块中，但这并不是必需的，因为渐变永远不会平白无故地直接绘制。

❷ 每个渐变都使用了三种颜色，但有四个结点，这是为了在最后循环到初始颜色。

❸ 两个矩形均占据整个图形七分之六的宽度（85.714%），且相互偏移了图形宽度的七分之一（14.286%）（矩形宽度的六分之一），这是为了对准渐变中色相大致相同的部分。注意由于 sRGB 模型的调整导致色相不会完美对齐，例如品红和青色中间仍然是浅紫色。

如图 6-8 所示，定义渲染服务的标记有时会比定义被渲染图形的标记还长。为了更好的组织结构，可重复使用的内容通常放置在文件的顶部，且通常使用 <defs>（definitions）元素来进行分组。另外，如果渐变只被使用一次，你可能希望直接在使用它的形状定义之前定义它。



为了获得理想的浏览器支持，应该在定义使用渐变的元素之前先在文档中定义渐变。许多版本的 Safari 将不能正常使用定义在文件后面的渐变。

使用 CSS 函数来定义渐变

渐变通过 Image Values and Replaced Content Module 规范被引入 CSS3 中，该规范是 W3C 候选提议且它在所有最新的 Web 浏览器中都得到了支持。如果想要支持旧的浏览器，需要给样式声明添加旧的实验性的语法前缀，但在 IE10 之前和 Opera Mini 浏览器中都不支持。

虽然 CSS 和 SVG 渐变最终的结果非常相似，但在定义和使用过程中还是有一些本质上的不同。另外，还有许多在细节上容易被忽略掉的差异，我们将在接下来的几章中重点介绍。

两者之间最明显的不同是定义的方式：SVG 渐变是通过 XML 标签元素来创建，而 CSS 渐变是在样式表内使用函数来创建。

使用 CSS 中的 `linear-gradient` 函数创建一个简单的线性渐变，函数的参数是使用逗号分隔的颜色结点值列表。每一个结点由空格分隔的颜色值和偏移值组成，如下所示：

```
background: linear-gradient(red 0%, lightSkyBlue 100%);
background: linear-gradient(#F0F 0%, #0FF 33.33%,
                            #FF0 66.67%, #F0F 100%);
```

CSS 线性渐变默认的方向是从上到下，而 SVG 默认的方向是从左到右（我们将在第 7 章中介绍如何改变两种类型渐变的方向）。

如果没有设置偏移值，结点将会在 0% 到 100% 之间平均分布。因此下列代码实现与上面代码相同的渐变效果：

```
background: linear-gradient(red, lightSkyBlue);
background: linear-gradient(#F0F, #0FF, #FF0, #F0F);
```

如果只指定了部分偏移的值，没有指定的值将在设置的点之间平均分配。

定义 CSS 渐变偏移时不能使用小数来代替百分数，但是你可以使用长度单位或 `calc()` 函数来定义它。例如，如下的 CSS 创建了一个背景渐变，在 `padding` 区域从黑色渐变到白色，但是无论元素多高，剩余部分范围内都是纯白色：

```
padding: 1em;
background: linear-gradient(black, white 1em,
                      white calc(100% - 1em), black);
```

你还可以给偏移设置大于 100% 或小于 0 的值，颜色将会根据渐变的可见部分是否是更大的渐变的一部分来进行调整。

在 CSS 渐变中没有办法直接设置 `stop-opacity` 值，但是你可以使用 `rgba()` 和 `hsla()` 颜色函数来定义半透明的颜色。

CSS 渐变和 SVG 渐变之间一个更细微不同是，CSS 不可以把渲染服务定义为一种唯一的资源类型，相反，CSS 渐变被定义为一个单独的没有固定尺寸和宽高比的矢量图形。

CSS 渐变函数用于代替 `url()` 对图片文件的引用，主要用在 `background-image` 列表（或者简写的 `background`）中。它还可以被用在 `list-style-image` 或 `border-image` 属性中，尽管浏览器是在支持背景渐变之后才支持这些属性的。

目前 CSS 渐变不可以用于代替 SVG 元素中 `fill` 和 `stroke` 属性引用的渲染服务，但在 SVG 2 中这一点很可能会改变。

第 7 章

各种形状和尺寸的渐变

目前为止，我们已经讨论了如何创建渐变，如何在渐变的起点（左）和终点（右）之间设置含有颜色和不透明度的结点的位置。如果你涉及的所有图形中渐变的方向都要求从左到右，那么现在所学的知识就已经足够了。但是，很可能你希望渐变是沿从上到下，或者 60 度的方向，或有其他的一些变换。

本章讨论了两种控制线性渐变位置的方式，还讨论了如何控制它的缩放。我们假定你已经可以正确使用基础的 SVG 形状元素（尤其是 `<line>`）、坐标系以及变换。它们之间细微复杂的区别你要心中有数，此外 SVG 和 CSS 在细节上也有一些区别，我们也会着重介绍这些内容。

不同的 Web 浏览器在平滑渲染渐变的效果上有明显差异，我们对此无能为力。尤其是在使用锐利的颜色过渡时，所以一定要测试代码，确保运行的结果你可以接受。

7.1　渐变矢量

设置渐变位置的第一种方式是使用 `x1`、`y1`、`x2` 和 `y2` 等属性来定位渐变的起点和终点，就像画一条线一样。默认情况下，`x2` 的值是 100%，其他属性的值为 0。如果你使用这些属性值来画一条线，它将沿着图形的顶部从左到右绘制：

```
<line x1="0" y1="0" x2="100%" y2="0" />
```

如果你想画一条从左上角到右下角的对角线，你将使用如下代码：

```
<line x1="0" y1="0" x2="100%" y2="100%" />
```

那么设置相同属性的线性渐变表示的是什么呢？这些由属性定义的线被称为渐变矢量。结点的偏移，以及它们之间颜色的混合值，都是根据这条线的长度来度量的。颜色会无限延伸到线条的两端。对于如下渐变，蓝色的结点在左上角（使用 x1 和 y1 定义的点），而绿色结点在右下角（使用 x2 和 y2 定义）：

```
<linearGradient id="blue-green"
                x1="0" y1="0" x2="100%" y2="100%" >
    <stop stop-color="blue" offset="0"/>
    <stop stop-color="darkSeaGreen" offset="1"/>
</linearGradient>
```

当然你也可以利用各个属性的默认值，仅使用以下属性就能实现与上述相同的渐变：

```
<linearGradient id="blue-green" y2="100%" >
```

如图 7-1 所示，设置渐变矢量属性值为 0% 或 100% 的所有可能的组合来填充长方形，去除起始点和终点为同一点的情况。每一列分别以不同角处的点（以 (*x1*,*y1*) 值定义）作为矢量的起点，每一行分别以不同的点作为终点。

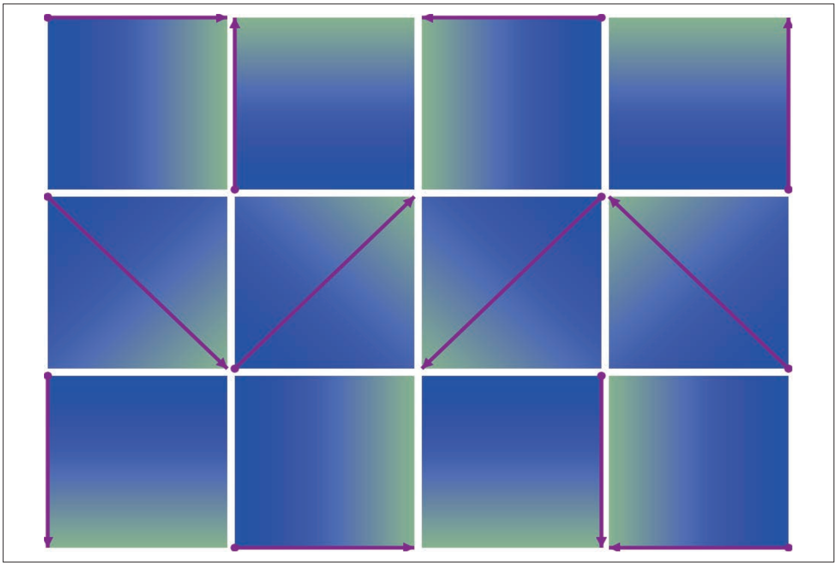


图 7-1：使用不同渐变矢量定义的渐变

如果渐变矢量的起点和终点相同，图形将会以纯色来进行填充。具体是哪一种颜色目前完全依赖于浏览器。SVG 规范中规定整个区域应该使用最后一个渐变结点的颜色，但是编写本书时，Blink 和 WebKit 浏览器以及 IE 浏览器都是用第一个渐变结点中定义的颜色来填充的。

例 7-1 给出了绘制图 7-1 的部分代码。代码中使用了渐变的简写形式：如果你在渐变中使用 `xlink:href` 属性来引用另一个渐变，新的渐变继承旧的渐变，然后根据新的属性来做出相应的改变。如本例所示，如果新的渐变没有任何 `<stop>` 元素，则会使用引用的渐变中的结点。

`xlink:href` 属性值是一个 URL，但是根据 SVG 1.1 规范，它应该指向相同文档中的一个元素。但是 Firefox 中支持链接外部文件中定义的渐变。

例 7-1 中在每个渐变的上方都绘制一条线来表示渐变矢量，`marker` 用于在线条的终点绘制箭头来告诉你它指向哪个方向。

例 7-1　使用属性来设置渐变矢量的位置

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="400px" height="300px" viewBox="0 0 400 300">
  <title>Same Gradient, Different Vectors</title>
  <style type="text/css">
      line {
          stroke: darkMagenta;                          ❶
          stroke-width: 2;
          marker-start: url(#start);
          marker-end: url(#end);
      }
      marker {
          fill: darkMagenta;
          stroke: none;                                 ❷
      }
      svg {
          overflow: visible;
      }
  </style>
  <defs>
    <linearGradient id="blue-green" >                   ❸
        <stop stop-color="blue" offset="0"/>
        <stop stop-color="darkSeaGreen" offset="1"/>
    </linearGradient>
```

```
    <marker id="start" viewBox="-2 -2 4 4">          ❹
      <circle r="1.5"/>
    </marker>
    <marker id="end" viewBox="-4 -2 4 4" orient="auto">
      <polygon points="-4,-2 0,0 -4,2"/>
    </marker>
  </defs>

  <g transform="translate(2,2)">                    ❺
    <svg width="1in" height="1in">                  ❻
      <linearGradient id="blue-green-1" xlink:href="#blue-green"/>
      <rect width="100%" height="100%" fill="url(#blue-green-1)"/>
      <line x1="0" y1="0" x2="100%" y2="0"/>        ❼
    </svg>

    <svg width="1in" height="1in" x="0" y="100">
      <linearGradient id="blue-green-2" xlink:href="#blue-green"
                      x2="100%" y2="100%"/>         ❽
      <rect width="100%" height="100%" fill="url(#blue-green-2)"/>
      <line x1="0" y1="0" x2="100%" y2="100%"/>
    </svg>

    <!-- 9 more samples omitted for length -->      ❾

    <svg width="1in" height="1in" x="300" y="200">
      <linearGradient id="blue-green-12"  xlink:href="#blue-green"
                      x1="100%" y1="100%" x2="0" y2="100%"/>    ❿
      <rect width="100%" height="100%" fill="url(#blue-green-12)"/>
      <line x1="100%" y1="100%" x2="0" y2="100%"/>
    </svg>
  </g>
</svg>
```

❶ `<line>` 元素必须设置 `stroke` 属性来变得可见，两个 `marker` 属性用于创建箭头效果。

❷ `marker` 使用与 `line` 描边相配的颜色填充，明确把 `stroke` 属性设置为 none 是为了解决 IE 浏览器中 `marker` 会从 `line` 继承样式的 bug。

❸ 第一个渐变永远都不会直接使用，它为所有其他的渐变定义渐变结点。

❹ `<marker>` 元素用于绘制箭头和矢量的轴点。

❺ 每一样块都绘制为 1 平方英寸（96px），为了添加内边距的效果，每隔 100px 放置一个样块，并从距离顶部和左边 2px 偏移的地方开始绘制。

❻ 每一个 `<linearGradient>` 使用 `xlink:href` 来从主渐变中复制结点，`blue-green-1` 渐变使用默认的位置属性来创建一个从左到右的渐变。

❼ 明确地使用相同的位置属性声明 `<line>` 元素来显示矢量。

❽ 对于第二个（以及以后的）样块，必须明确设置渐变矢量的部分或全部位置属性。

❾ 你应该可以从图 7-1 所示的箭头想象出省略的代码是什么，把它们都打印在书上就显得太啰嗦了。更好的做法是使用 JavaScript 来生成所有的元素而不是重复标记。

❿ 对于最后一个渐变，所有的位置属性都被明确设定来创建一个沿着长方形底部从右到左的渐变矢量。注意最终生成的渐变在效果上与沿着长方形顶部从右到左的矢量创建的渐变完全相同。

图 7-1 中的所有渐变例子都是从一个角到另一个角，跨越整个正方形。但并不是必须这样做。如果矢量没有覆盖整个形状，默认会使用纯色来填充剩下的部分。效果就好像分别给第一个和最后一个结点设置偏移使它们偏离渐变矢量的起点和终点。

> 改变渐变矢量的长度和改变偏移在重复渐变中会有不同的效果，我们将在第 8 章中对此进行讨论。

渐变矢量的起点和终点可以扩展到 0% 到 100% 这个区域之外，这与结点偏移不同，不会位于 0% 到 100% 之间。你可能会问："100% 是相对于谁的呢？"

7.2 对象边界盒

在例 7-1 中，我们使用嵌套的 `<svg>` 元素在每个占据 100% 宽和高的长方形（也就是每个渐变）中创建局部坐标系。这意味着可以给 `<line>` 元素的坐标设置百分比，并让它直接匹配渐变向量使用的百分比。

通常你要使用渐变填充的形状不会占据整个坐标系统。所以，理解这一点非常重要：渐变矢量属性中的值默认并不是根据绘制 `<line>` 时使用的局部坐标系来度量的。相反，它们是基于要填充的形状的对象边界盒创建的新坐标系来定义的。

那么什么是对象边界盒呢？对于像例 7-1 中使用的长方形，它的对象边界盒就是长方形。更准确地说，就是长方形本身的几何面积且不包括任何描边。对于其他形状来说，就是可以将图形包含在本身（可能是变换后）坐标系内的最小矩形。再次声明，边界盒不包括任何描边或标记。

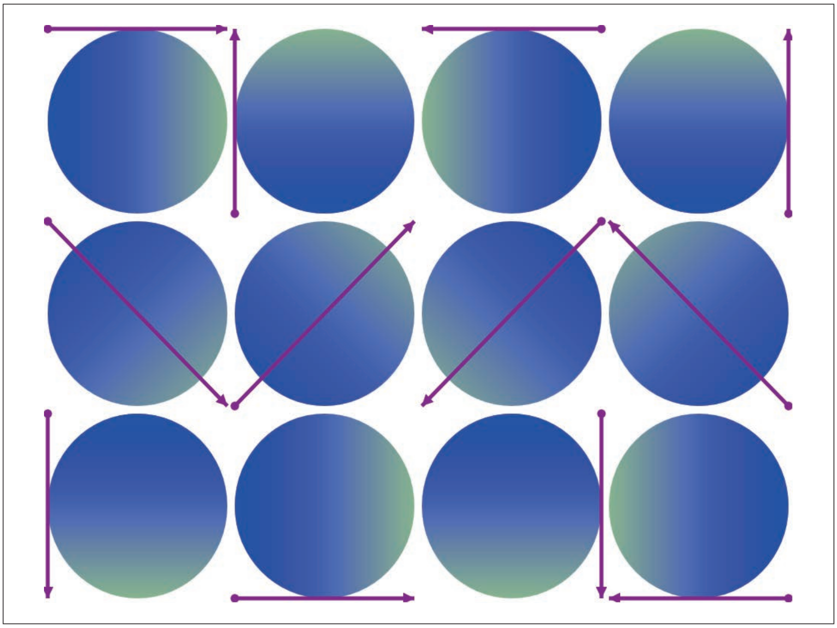


图 7-2：使用不同渐变矢量定义的渐变来填充圆



如果形状在去除描边或标记之后高或宽为 0，那么它的边界盒为空，且渐变将会视为错误来处理（将会使用降级的颜色）。这在绘制直线时可能会有问题，我们将在第 13 章中讨论解决办法。

对于非矩形形状，渐变矢量的起点和终点可能在形状的外部，也可能在 0% 和 100% 范围之内。图 7-2 显示了用圆来替换例 7-1 中所有长方形的最终效果：

```
<circle cx="50%" cy="50%" r="50%" fill="url(#gradient-id)"/>
```

每个圆都尽可能地填满它的局部坐标系，但四个角无法填充。渐变的定义方式没有改变，所以渐变矢量依然是从角到角的。对角矢量的最终结果是渐变中起始和终止的颜色结点被裁剪掉了。重新审视我们的渲染服务，渲染效果被缩放到适应整个对象边界盒，不过后来遮罩带被删除露出下面形状。

如上所述，对象边界盒整体是一个全新的坐标系。它不仅仅把百分比缩放到适合整个盒子，而且对基本的用户单位也做了同样的处理。水平方向上

的一个单位等于边界框宽度的 100%，垂直方向上的一个单位等于边界框高度的 100%。换而言之，你可以使用 0 到 1 之间的小数来代替百分比。

你甚至还可以使用带单位的长度值，但是（与 SVG 中其他形式的缩放一致）长度单位也会随着用户单位来缩放。1px 的长度等于 1 个用户单位、等于新坐标系中宽或高的 100%。所有其他长度单位将等于形状宽或高的几倍。

因为单位被缩放到匹配边界盒的宽和高，所以 1 水平单位不一定等于 1 垂直单位。例 7-1 中它们相等是因为被填充的形状是一个正方形。最终，对角渐变矢量是一条完美的沿 45 度方向的线。这也意味着与渐变矢量垂直（成直角）的线是一条相对方向的 45 度线，即在相对的角之间的对角线。

这一点非常重要，因为这些垂直线相当于拥有相同颜色的渐变的一部分。结点颜色值将会沿着渐变矢量垂直线向两侧扩展。但渐变矢量正好在连接两个相对角的盒对角线上时，渐变的中点（等于 50% 的节点偏移）的垂直线沿着盒子剩下两个角之间的对角线延长。例 7-2 构建了一个在 50% 偏移的位置发生颜色骤变的渐变来突出这种关系。

例 7-2　通过对角渐变创建倾斜的条纹

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%">
    <title xml:lang="en">Stretched Object Bounding Box gradients
    </title>
    <style type='text/css'>
    </style>
    <linearGradient id="diagonal" y2="100%">
        <stop offset="0" stop-color="white"/>
        <stop offset="0.5" stop-color="red"/>
        <stop offset="0.5" stop-color="blue"/>
        <stop offset="1" stop-color="white"/>
    </linearGradient>
    <rect height="100%" width="100%" fill="url(#diagonal)"/>
</svg>
```

在例 7-2 中，作为整体的 SVG 和渐变填充的长方形都被设置为 100% 来填充浏览器窗口或使用 SVG 绘制的图片区域的宽和高。图 7-3 显示了对角渐变绘制在一个正方形区域时的效果。渐变矢量是从左上到右下的 45 度的对角线，渐变 50% 偏移位置的条纹正好是右上到左下的对角线。

如果你要填充的形状（更准确地说，是对象边界盒）不是一个完美的正方形，将会发生什么呢？对角线将不是 45 度角。但是，由于对象边界盒坐标系的拉伸效果，渐变的中点依然在相对角的对角线上，如图 7-4 所示。

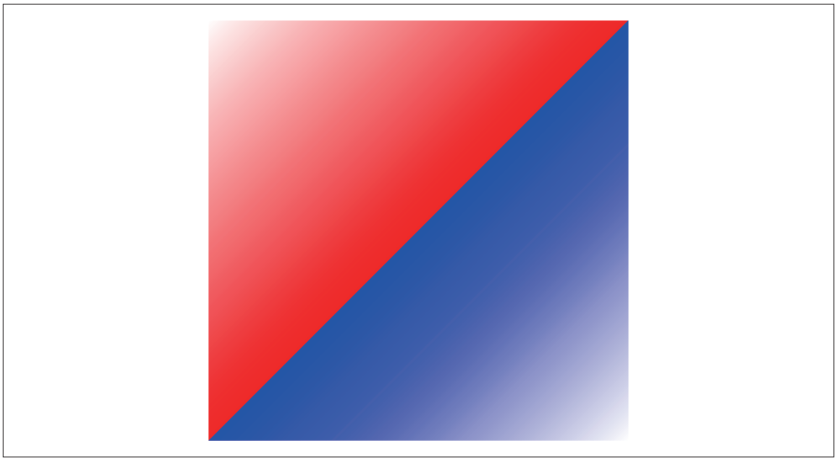


图 7-3：在一个正方形中，绘制一个在 50% 偏移的位置发生颜色骤变的对角渐变

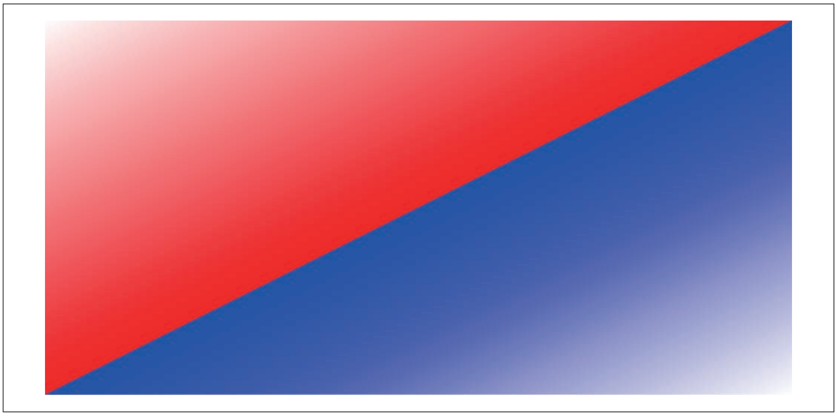


图 7-4：在一个长方形中，绘制一个在 50% 偏移的位置发生颜色骤变的对角渐变

颜色相同的线不再垂直于我们所说的一般意义上的渐变矢量，至少在坐标系表象中不是这样。在拉伸的坐标系中，它们和垂直线在 x 和 y 坐标之间拥有相同的数学关系。在数学上，这种“理论垂直”线实际上被称为法向量，但是数学家对什么是法向量有不同的见解。

对象边界盒坐标系因此创建了一个不均匀的缩放，类似于有两个参数的 `scale(sx,sy)` 变换或 `preserveAspectRatio="none"` 设置。

7.3 在盒子表面绘制

如果你阅读仔细，你可能注意到了不久前出现的“默认”。对象边界盒单位是定义渐变矢量的默认值，但它不是唯一的选择。你可以在 <linearGradient> 元素上使用 gradientUnits 属性来改变它。

设置 gradientUnits 的值为 userSpaceOnUse 使浏览器在用于绘制要填充的形状的坐标系内解析你的 x1、y1、x2 和 y2 属性。

> 如果你想明确地设置默认行为，gradientUnits 的另一个值是 objectBoundingBox。

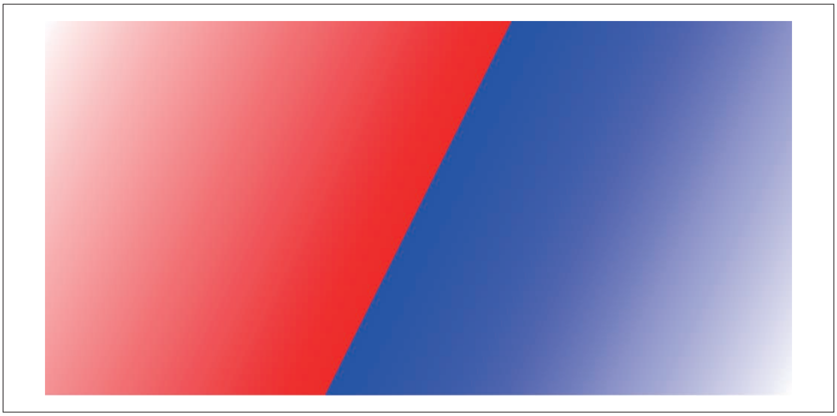


图 7-5：在一个完全填充其坐标系长方形中，绘制一个在 50% 偏移位置发生颜色骤变的用户空间对角渐变

图 7-5 显示了修改例 7-2 中 gradientUnits 属性值的结果：

```
<linearGradient id="diagonal" y2="100%"
                gradientUnits="userSpaceOnUse">
```

由于矩形完全填充了坐标系，渐变矢量仍然从其左上角指向右下角。然而，由于坐标系本身没有扭曲，50% 的结点沿着真实世界的直角对角绘制，而不再连接另外两个角。

避免坐标系被扭曲仅仅是使用 userSpaceOnUse 的一个原因。通过 userSpaceOnUse 单位，你可以使单个渐变在形状与形状之间效果一致。实例见例 7-3，最终

结果如图 7-6 所示。

例 7-3 使用 userSpaceOnUse 渐变单位来填充多个形状

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="400px" height="300px" viewBox="0 0 400 300">
    <title xml:lang="en">Tropical Sunset User-Space Gradient</title>
    <linearGradient id="sunset" gradientUnits="userSpaceOnUse"
                    y1="1em" x2="0" y2="250px">                  ❶
        <stop stop-color="midnightBlue" offset="0"/>
        <stop stop-color="deepSkyBlue" offset="0.25"/>
        <stop stop-color="yellow" offset="0.5"/>
        <stop stop-color="lightPink" offset="0.8"/>
        <stop stop-color="darkMagenta" offset="0.99"/>
        <stop stop-color="#046" offset="0.99"/>                  ❷
    </linearGradient>

    <rect height="100%" width="100%" fill="dimGray" />           ❸

    <g fill="url(#sunset)">                                      ❹
        <rect x="20" y="20" width="100" height="120" />
        <rect x="280" y="20" width="100" height="120" />
        <rect x="20" y="160" width="100" height="120" />
        <rect x="280" y="160" width="100" height="120" />
    </g>
    <rect x="140" y="0" width="120" height="300"
          fill="url(#sunset)" />                                 ❺
</svg>
```

❶ 给 `linearGradient` 设置 `userSpaceOnUse` 值允许我们使用长度（例如 1em 或 250px）来定位渐变矢量。`x1` 属性没有指定，所以使用默认值 0，而 `x2` 属性（默认为 100%）被重置为 0 来创建一个直线垂直渐变。

❷ 最后两个渐变结点拥有相同的偏移值，在夕阳的天空和静止的水面之间创建了一条分界线。所有超出渐变矢量范围的点都会被渲染为暗蓝绿色 `#046`。

❸ 在使用渐变填充的元素下，放置一个灰色的长方形来填充整个 SVG。

❹ 与往常一样，你可以在组元素上定义一个填充值，它会被所有的子形状继承。但是，每个形状仍然会单独计算渐变，如果使用的是 `objectBoundingBox` 渐变，每个形状的渐变将会基于它自己的边界盒，而不是组。

❺ 为了证明渐变的连续性与组无关，中间的长方形单独进行绘制。它填满了 SVG 的高度，所以你会看到整个渐变，包括超出渐变矢量的纯色部分。

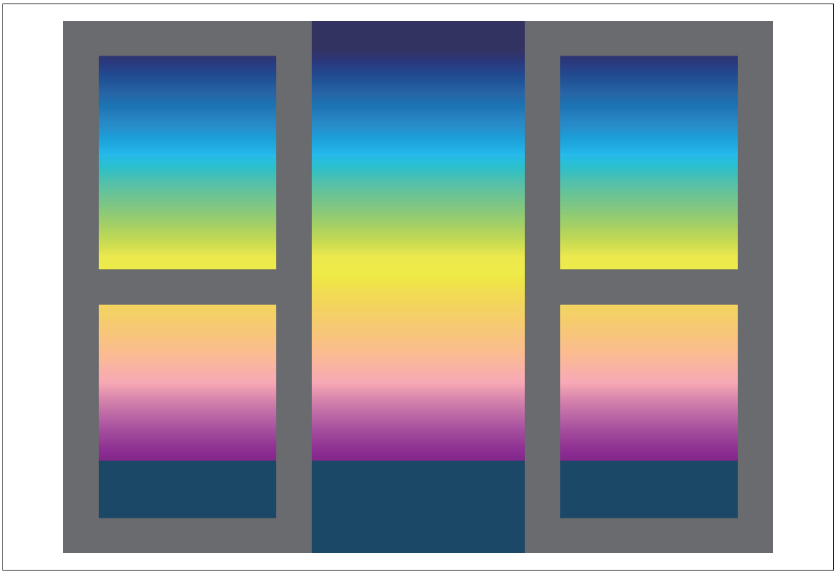


图 7-6：连续跨越多个图形的渐变

在例 7-3 初始代码的最后，`offset="1"` 处有锐利的渐变变换（在天空和大海之间）。根据规范，当多个结点拥有相同的偏移值时，该点以及超出部分填补的颜色应该使用最后一个结点中定义的颜色。然而，Blink 和 WebKit 浏览器会忽略最后一个结点，而使用 `darkMagenta` 来填充剩下的空间。

为了使浏览器之间有相同的结果，避免在 0% 和 100% 的偏移处使用锐利的过渡。

图 7-6 中的图片可以使用单个连续渐变来填充背景，然后通过复杂的路径来在夕阳之上创建开着的阳台门形状。相反，灰色门框实际上是背景唯一可见的地方，窗户和开放区域其实是在它上面绘制的长方形。它们外观一致是因为它们都是用相同的连续渐变来填充。但是，请记住“用户空间”依然不是一个绝对的坐标系，它会被任何变换或应用在填充形状的嵌套坐标系所影响。

所有浏览器都会把变换应用到用户空间渲染服务上，但嵌套的坐标系更加古怪。WebKit、Blink 和 IE 使用渐变的父级 SVG，而不是形状，来把百分比转换为用户单位。

CSS 与 SVG
使用关键词来设置 CSS 渐变的位置

正如第 6 章中说的那样，CSS 线性渐变的默认方向是从上到下。当然你也可以改变它的默认方向。它的语法与 SVG 完全不同，且与早期试验性的 CSS 渐变也有相当多的不同；为了尽可能地支持早期的浏览器，你可能需要使用一个 CSS 预处理程序或脚本来把你的渐变重构为旧的语法。

要想改变 CSS 渐变的方向，需要在函数中颜色结点之前添加一个额外的参数。要创建整整齐齐从一边到另一边、从一个角到另一个角的渐变，你可以使用关键词：第一个关键词是 `to`，接着是 `left`、`right`、`top` 或 `bottom` 之一来表示边，也可以是这些关键词的组合（例如 `top right`）表示角。这些关键词描述的是渐变的终点（也就是渐变要往哪个方向），起点在相反一侧或相对的角。

当使用角来表示时，渐变过渡的角度会被调整，所以渐变 50% 偏移位置点连接着另外的角。虽然规范中实现的方式不同，但它的效果和对象边界盒单位中角到角的 SVG 渐变效果一样。

例 7-4 直接比较了例 7-2 中 SVG 对角渐变（作为图片嵌入）和一个相似的 CSS 渐变（在网页的 `body` 上）。CSS 渐变中的 `to bottom left`，意思是它起始于右上角。图 7-7 显示了最终效果，包括内嵌的 SVG 渐变。

例 7-4　使用 CSS 创建对角条纹渐变

```
<!DOCTYPE html>
<html xml:lang="en">
<head>
    <meta charset="utf-8" />
    <title>Positioning CSS gradients using keywords</title>
    <style type='text/css'>
    html, body {
        height: 100%;
        margin: 0; padding: 0;
    }
    body {
        background: linear-gradient(to bottom left,
                        white, red 50%, blue 50%, white);
    }
    img {
        position: absolute;
        width: 50%; height: 50%; l
        eft: 25%; top: 25%;
    }
```

```
    </style>
</head>
<body>
    <img src="gradient-vector-diagonals.svg">
</body>
</html>
```

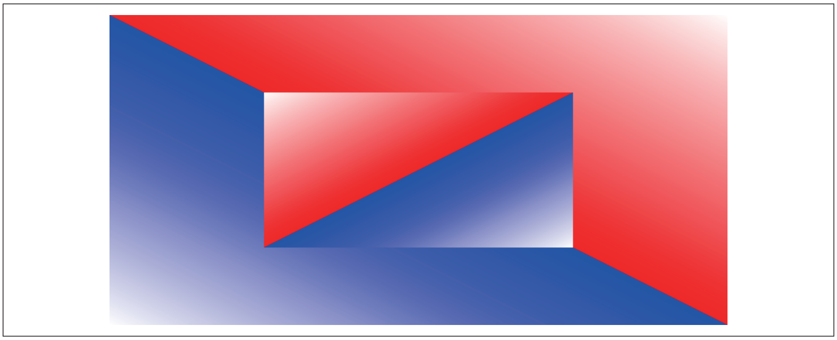


图 7-7：把一个 CSS 渐变作为一个 SVG 渐变的背景



这个以及其他 CSS 渐变都是 Firefox 中的截图。编写本书时，Blink 浏览器在渲染 CSS 渐变中沿着对角线的急剧转换，尤其是像本例中这种大型背景时，性能很差，对角线会参差不齐。这个问题在 SVG 渐变中有所消减，但并没有完全消除。

对于较小的图片，通过一两个百分比点来分隔结点偏移能取得可接受的结果 ，这样就强制创建一个抗混叠的效果。但像这样整个页面的渐变，只能创造锯齿状的紫色线。

在 CSS 中，你无法控制渐变矢量的长度，线性渐变通常会被从角到角调整到刚好适合图片区域。如果要在内容两侧创建纯色块，或扩展渐变溢出盒子，你可以通过调整第一个和最后一个结点的偏移值来实现。在讨论 SVG 渐变时我们曾提过，调整渐变矢量长度和调整偏移的值之间的不同对重复渐变很重要。

CSS 渐变总是创建长方形图片，所以它没有复杂的对象边界盒。然而，如果你使用圆角边框或裁剪来减少图片的可见区域，这将裁减掉渐变中类似于 SVG 中的圆和其他图形的角。

CSS 中没有直接对应用户空间渐变的定义。对于背景图片，你可以使用 `background-attachment:fixed` 来实现相似的效果。

7.4　渐变，变换

本章的开头我们提到过 SVG 中有两种控制线性渐变方向的方式。第二种方式是使用 `gradientTransform` 属性。

`gradientTransform` 属性的值是可以控制形状位置和方向的相同变换函数的列表：

- `translate(tx,ty)` 仅改变位置而不会扭曲
- `scale(s)` 或 `scale(sx,sy)` 放大、缩小、拉伸或翻转成镜像
- `rotate(a)` 或 `rotate(a,cx,cy)` 围绕原点或指定点进行旋转
- `skewX(a)` 或 `skewY(a)` 沿着轴或其他线倾斜

可以串联列出任意多个这些函数。

聚焦未来

使用 CSS 规则来变换渐变

CSS Transforms Module 把 `gradientTransform` 这个表现属性映射到了新的 CSS 样式变换属性。虽然目前还没有浏览器支持，但在未来，你将可以在渐变元素上设置变换样式，其效果会与 `gradientTransform` 属性相同。

CSS 变换的语法与 SVG 中稍微有些不同，CSS 中长度值和角度值需要设置单位，但 SVG 1.1 中这些值通常定义为数字。新模块中引入了新的简写的变换函数，例如 `translateY` 和 `scaleX`；还废弃了三个值的旋转函数（CSS 中不能使用），取而代之的是一个单独的 `transform-origin` 属性，它将会影响渐变变换的效果。

CSS Transforms Module 还引入了三维变换函数。但它们不适用于渲染服务，并且会导致整个变换列表失效。

通过变换线性渐变创建的效果与操作渐变矢量创建的效果类似。对于任意给定的图形，它们中总有一个更加易于计算。

> 你可以结合使用位置属性和变换。变换形状时，位置属性可以在变换坐标系中计算。

变换一个渐变是什么意思呢？它的意思是渲染服务创建的墙纸在裁剪为要填充的形状之前进行了变换。相比之下，当你变换一个使用渐变填充的形状时，形状和渐变都会被变换。

图 7-8 比较了在变换和未变换的形状中使用变换和未变换渐变的两种效果。例 7-5 是对应的代码。

例 7-5　变换中的形状和渐变

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="400" height="500" viewBox="0,0 400,500">
    <title xml:lang="en">Transforming shapes, gradients, or both
    </title>

    <linearGradient id="stripe" x2="0" y2="100%">          ❶
        <stop offset="0" stop-color="yellow"/>
        <stop offset="0.1" stop-color="gold"/>
        <stop offset="0.5" stop-color="tomato"/>
        <stop offset="0.5" stop-color="blueViolet"/>
        <stop offset="0.9" stop-color="indigo"/>
        <stop offset="1" stop-color="midnightblue"/>
    </linearGradient>
    <linearGradient id="stripe-transformed" xlink:href="#stripe"
                    gradientTransform="skewY(25)" />       ❷

    <g fill="url(#stripe)" >                               ❸
        <rect height="190" width="190" x="5" y="5" />
        <rect height="190" width="190" x="5" y="5"
              transform="translate(200,0) skewY(25)"/>     ❹
    </g>
    <g transform="translate(0,200)"
       fill="url(#stripe-transformed)">
        <rect height="190" width="190" x="5" y="5" />      ❺
        <rect height="190" width="190" x="5" y="5"
              transform="translate(200,0) skewY(25)"/>     ❻
    </g>
</svg>
```

❶ 基础渐变使用位置属性来定义从上到下的方向：起始点是默认的（0,0），终点是（0,100%）。

❷ 第二个渐变使用 `xlink:href` 属性来复制相同的颜色结点图案，`gradientTransform` 属性用于把图案进行倾斜变换。

❸ 前两个 `<rect>` 元素被分为一组来使用相同的填充值。它们都会使用未变换的渐变来填充。

❹ 第二个正方形先水平移到图形的右上部分，然后再倾斜。但它填充所使用的渐变本身没有倾斜，所以颜色结点依然完全与正方形的顶部和底部边缘对齐。

❺ 下面一行的正方形使用了变换之后的渐变，左边的正方形本身没有倾斜，而只倾斜了渐变。

❻ 最后一个正方形本身是倾斜的，且使用倾斜的渐变填充。所以最终渐变角度是混合之后的效果。

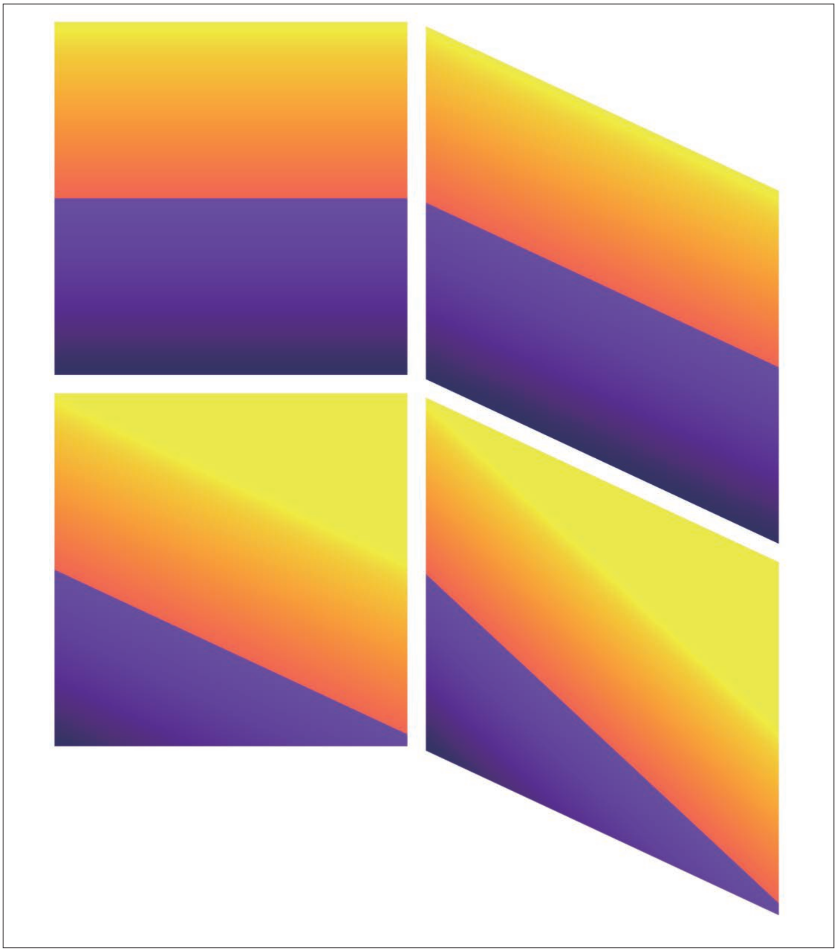


图 7-8：倾斜渐变和正方形：未变换的渐变（上）和倾斜后的渐变（下）

形状的变换会维持渐变颜色和形状边缘之间的关系：图 7-8 中同一行的正方形对应的角都有相同的颜色，而不管形状整体是否被倾斜。相比之下，当渐变被变换时，它会独立于形状变换，改变渐变的可见部分。`midnightblue` 颜色几乎完全被切掉，而纯黄色则大部分是可见的。

没有直接的方式可以只变换形状而不变换填充它的渲染服务。即使是 `userSpaceOnUse` 渐变，也会在使用它的形状的坐标系统变换中反映出来。

> 如果要在变换的形状内使用未变换的渲染服务，可以使用裁剪路径来使变换形状的边界正好放在一个更大的、使用渲染服务填充的未变换的长方形内。

倾斜显然是渐变变换来创建对角渐变的一种方式。旋转是另一种方式。例 7-6 创建了彩虹渐变并将它间隔 45 度进行旋转，之后用生成的渐变分别填充 SVG 边和角周围的椭圆。最终结果如图 7-9 所示。

例 7-6　使用旋转变换渐变

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="400" height="300" viewBox="-200,-150 400,300">   ❶
    <title xml:lang="en">Rotated Gradients</title>
    <defs>
        <linearGradient id="rainbow" >
            <stop stop-color="darkViolet" offset="0"/>
            <stop stop-color="blue" offset="0.143"/>             ❷
            <stop stop-color="cyan" offset="0.286"/>
            <stop stop-color="limeGreen" offset="0.429"/>
            <stop stop-color="yellow" offset="0.572"/>
            <stop stop-color="orange" offset="0.715"/>
            <stop stop-color="red" offset="0.857"/>
            <stop stop-color="maroon" offset="1"/>
        </linearGradient>
        <linearGradient id="rainbow45" xlink:href="#rainbow"
                        gradientTransform="rotate(45)"/>        ❸
        <linearGradient id="rainbow90" xlink:href="#rainbow"
                        gradientTransform="rotate(90)"/>
        <linearGradient id="rainbow135" xlink:href="#rainbow"
                        gradientTransform="rotate(135)"/>
        <linearGradient id="rainbow180" xlink:href="#rainbow"
                        gradientTransform="rotate(180)"/>
        <linearGradient id="rainbow225" xlink:href="#rainbow"
                        gradientTransform="rotate(225)"/>
        <linearGradient id="rainbow270" xlink:href="#rainbow"
                        gradientTransform="rotate(270)"/>
```

```
    <linearGradient id="rainbow315" xlink:href="#rainbow"
                    gradientTransform="rotate(315)"/>
  </defs>

  <ellipse rx="60" ry="40" cx="130" cy="0"
           fill="url(#rainbow)"/>                          ❹
  <ellipse rx="60" ry="40" cx="130" cy="100"
           fill="url(#rainbow45)"/>
  <ellipse rx="60" ry="40" cx="0" cy="100"
           fill="url(#rainbow90)"/>
  <ellipse rx="60" ry="40" cx="-130" cy="100"
           fill="url(#rainbow135)"/>
  <ellipse rx="60" ry="40" cx="-130" cy="0"
           fill="url(#rainbow180)"/>
  <ellipse rx="60" ry="40" cx="-130" cy="-100"
           fill="url(#rainbow225)"/>
  <ellipse rx="60" ry="40" cx="0" cy="-100"
           fill="url(#rainbow270)"/>
  <ellipse rx="60" ry="40" cx="130" cy="-100"
           fill="url(#rainbow315)"/>
</svg>
```

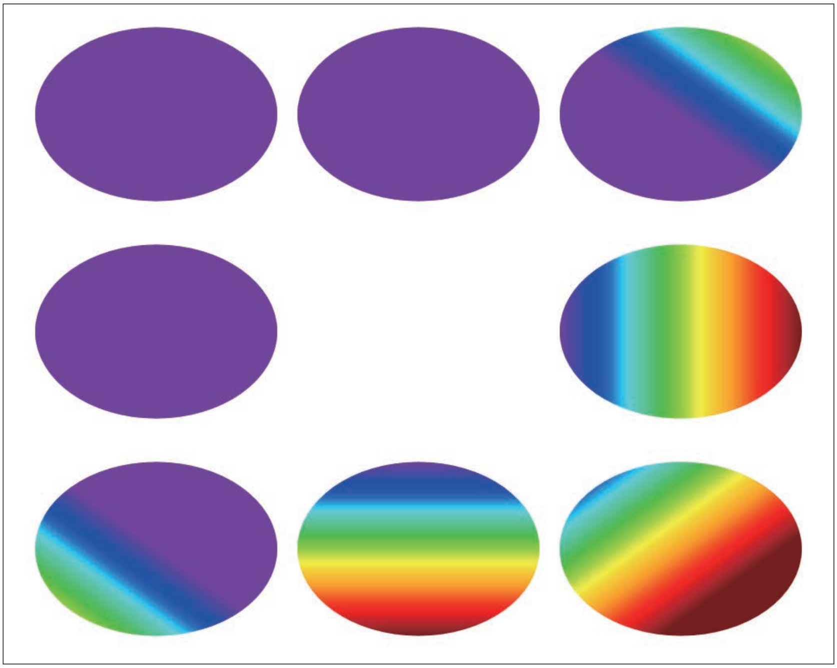


图 7-9：用旋转渐变填充的椭圆

❶ SVG 使用以原点为中心的坐标系。

❷ 彩虹渐变拥有八个结点，所以偏移都是 1/7（0.143）的倍数。

❸ 使用 `xlink:href` 来引用源渐变来生成几乎一样的渐变，然后给它添加新的属性：本例中是 `gradientTransform`。

❹ 源渐变使用默认方向从左到右排列；使用该渐变填充的椭圆的位置在 SVG 的右侧边上。剩余的椭圆围绕 SVG 按照顺时针的方向放置，并与沿顺时针方向依次增加的渐变变换角度相对应。

等等，为什么一些形状是使用纯紫罗兰色填充而不是使用渐变填充的？这是因为椭圆是围绕 SVG 坐标系的中心排列的，而渐变是围绕它们各自的对象边界盒坐标的中心来旋转的。正如所有的 SVG 坐标系一样，它的默认原点是左上角，但是渐变没有 `viewBox` 属性来改变它。

渐变矢量（初始状态下，沿着每个形状的边界盒的顶部从左到右）是围绕每个盒子的原点（左上角）来旋转的。把渐变矢量旋转 90 度（如底部中间的椭圆所示）将会使渐变从上到下改变。然而，再进一步旋转的话，渐变矢量将会指向形状边界盒的外部。形状的大部分或全部在渐变起始之前的区域中，所以会使用第一个渐变结点中定义的颜色（上述中的 `darkViolet`）渲染。

为了有效地在对象边界盒中使用线性渐变的旋转变换，你需要一些额外的操作。有以下可选方式。

把 `gradientTransform` 属性的值翻译为位置属性来在形状内重新定位渐变。

使用三个值的 `rotate(a, 0.5, 0.5)` 函数来使其围绕坐标系的中心点旋转（如果使用 CSS 规则，也可以使用 `transform-origin` 属性）。

改变矢量的位置属性，这样旋转后它就可以沿着原点正确的一侧。

还有更复杂的问题，尽管你已经变换了渐变的方向，但是旋转并不能改变它的长度，结点的偏移值依然是相对于渐变矢量水平长度来计算的。由于对象边界盒坐标的扭曲变形，水平渐变的默认长度（1 水平单位）在旋转 90 度之后，依然会填充垂直的盒子（1 个垂直单位）。然而，它将无法匹配旋转 45 度之后的对角长度（拉伸为 1.141 个单位），所以 50% 的偏移将不能正好连接盒子的另外两个角。

换句话说，渐变旋转的时候会拉伸或收缩，但可能不是按照你预期的方式。

如果你在渐变上使用 `userSpaceOnUse` 单位，它将会围绕整个 SVG 坐标系的

原点（本例中，图片的中心点）旋转。图 7-10 显示了这种效果。相对于例 7-6，代码中唯一的修改是主要渐变元素的属性：

```
<linearGradient id="rainbow" gradientUnits="userSpaceOnUse"
                   x1="20%" x2="50%">
```

`x1` 和 `x2` 的值必须改变，因为现在百分比将会根据 SVG 的宽度而不是形状的宽度来计算。这些新的属性只需要设置一次，它们会自动被复制到所有引用它的元素上。

如果你仔细看，图 7-9 和图 7-10 还有另外一个不同之处。为了看得更清楚一些，图 7-11 取出每个图形中右下角的椭圆（使用 `rotate(45)` 渐变变换的椭圆）。左侧的椭圆使用对象边界盒渐变，而右侧的椭圆使用的是用户空间坐标。它们的角度不完全相同。使用对象边界盒的渐变，45 度看起来更像旋转了 60 度。

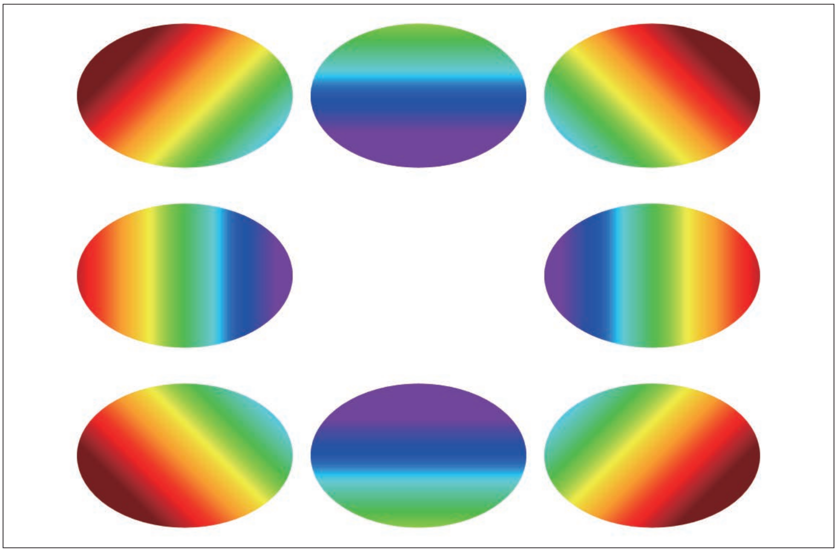


图 7-10：使用在用户空间旋转的渐变填充椭圆

这是因为变换角度是在对象边界盒创建的拉伸坐标系内计算的。如果长方形边界大致上是一个正方形，最终的角度将和你定义的角度一样。如果不是正方形，就像这些椭圆一样，角度将会相应地压缩或拉伸。对象边界盒的 45 度通常正好是盒子的对角线，但是在其他坐标系中，将不一定匹配相同的角度。

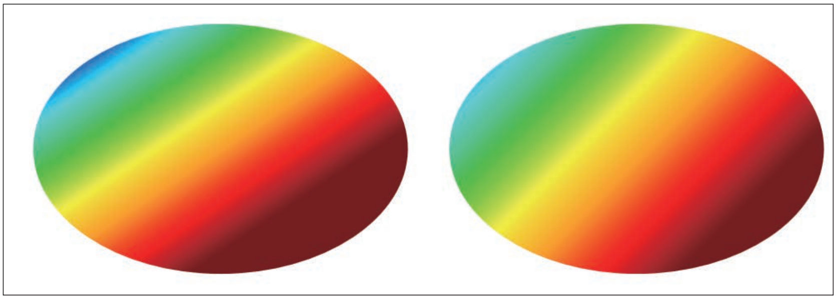


图 7-11：比较渐变在对象边界盒空间（左）和用户坐标空间（右）内旋转的结果

CSS 与 SVG

使用角度定位 CSS 渐变

CSS 渐变同样可以使用旋转角度来定位，但是结果与 SVG 中有所不同。

你可以使用带单位的角度作为 linear-gradient 函数的第一个参数，来代替方向关键词。由于默认方向是从上到下，角度是相对于朝下的垂直矢量来计算的，为了创建 SVG 默认的指向右侧的渐变，必须把它设置为 -90 度。

例 7-7 改编自例 7-4，它比较的不是 CSS 渐变和 SVG 渐变，而是两种 CSS 渐变。<body> 上的渐变使用的是 to 角的语法来倾斜，而内部的 <div> 元素上的渐变使用角度来倾斜。最终的网页如图 7-12 所示。

例 7-7　使用角度而非关键词来控制 CSS 渐变

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>Angling CSS gradients versus using corners</title>

    <style type='text/css'>
    html, body {
        height: 100%;
        margin: 0; padding: 0;
    }
    body {
        background: linear-gradient(to bottom left,
                        white, red 50%, blue 50%, white);
    }
    div {
        background: linear-gradient(-45deg,
```

```
                              white, red 50%, blue 50%, white);
            position: absolute;
            width: 50%; height: 50%;
            left: 25%; top: 25%;
        }
    </style>
</head>
<body>
    <div></div>
</body>
</html>
```

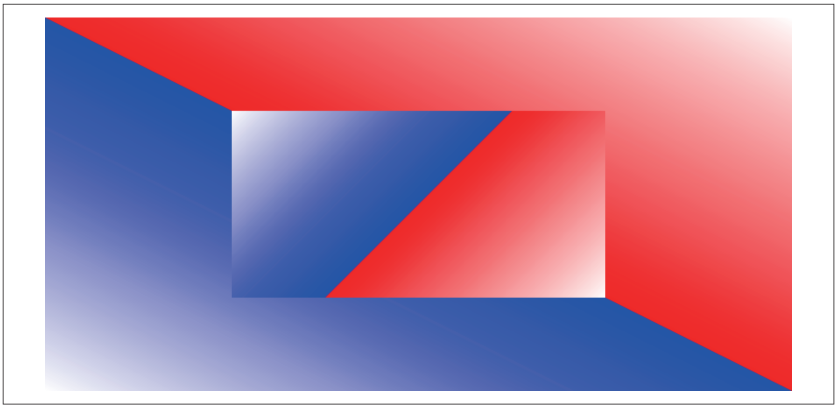


图 7-12：在角到角的 CSS 渐变上放置一个固定角度的渐变

使用角度定义的渐变会精确地使用定义的角度绘制，即使这意味着 50% 的结点不再连接元素的左下角和右上角。渐变矢量的长度和位置会自动调整，所以渐变依然从左上角开始到右下角结束。再次说明，创建不在角上起始和终止的 CSS 渐变的唯一方式是调整个别颜色结点的偏移值。

第 8 章

重复

在第 7 章中操作渐变矢量和渐变变换时，我们曾数次指出超出渐变矢量的区域会被填充为纯色，也提到这只是默认行为。

本章会探讨替代的方案——重复渐变，它变换尽可能多的次数来填充形状。同样，SVG 和 CSS 都可以实现这种效果，我们也会对两者进行对比。

基于现在已经介绍的线性渐变的所有属性，该章最后将介绍一些关于 SVG 渐变常见的一个用途（但不幸的是，也有很多问题）的例子：在 HTML 页面中设定重复使用的 SVG 图标的样式。如果想要达到预期结果，你需要避免 Web 浏览器中的一些 bug。

8.1 如何扩展渐变

超出渐变矢量终点部分的渐变外观可以通过 `<linearGradient>` 元素上 `spreadMethod` 属性的值来设置。它控制着渐变如何无限扩展。

该属性的默认值是 `pad`。矢量之前的部分都使用第一个 `stop-color` 的值来填补，而矢量终点之后的部分都使用最后一个 `stop-color` 的值来填补。我们之前看到的例子使用的都是默认的填补行为，例 8-1 中明确设置了该值，`x1` 和 `x2` 属性把渐变限制在对象边界盒中间的 10%，但是形状剩下的部分将会被蓝色和粉红色填补。渐变结果如图 8-1 所示。

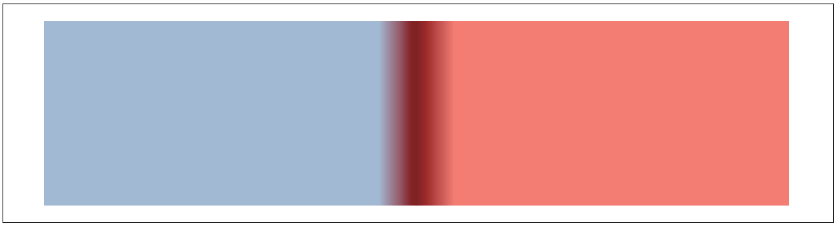


图 8-1：使用纯色填补的狭窄渐变

例 8-1　使用填补渐变来填充形状

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4in" height="1in">
    <title xml:lang="en">Padded Gradient</title>
    <linearGradient id="divider" spreadMethod="pad"
                    x1="45%" x2="55%" >                   ❶
        <stop stop-color="lightSteelBlue" offset="0%"/>
        <stop stop-color="darkRed" offset="50%"/>
        <stop stop-color="salmon" offset="100%"/>         ❷
    </linearGradient>
    <rect width="100%" height="100%" fill="url(#divider)" />
</svg>
```

❶ 你不需要设定 `spreadMethod` 的值，因为 `pad` 是默认值；这里是为了突出它。

❷ 注意结点的偏移值是渐变矢量长度的百分比，而不是对象边界盒或用户空间的百分比。

`spreadMethod` 属性的其他可选值有 `repeat` 和 `reflect`。这两个值都会导致渐变重复尽可能多的次数来填充形状。

WebKit（Safari 浏览器和旧版本的 Chrome）不支持 `spreadMethod` 属性的 `repeat` 和 `reflect` 值。Firefox 暂时在改变底层代码时不支持重复和映射的渐变，它们在 2014 年的 32 版本中再次引入。当不支持时，所有的渐变都按照填补颜色来渲染。

如果重复效果对于你的图形来说必不可少，你有两种解决办法：

- 尽可能多次地手动重复渐变结点（并调整渐变矢量的尺寸）来填充你的形状；
- 使用 `<pattern>` 元素（第 10 章中介绍）来创建重复渐变纹理。

8.2　无穷渐变映射

`reflect` 方法在每个重复周期内，都会反转结点的顺序。这会在起始点和终点之间有一个平滑的过渡，在定义的颜色交替时不会有任何不连续的感觉。颜色在波峰和波谷之间交替。

在例 8-1 的基础上将 `spreadMethod` 属性的值修改为 `reflect`，结果如图 8-2 所示。

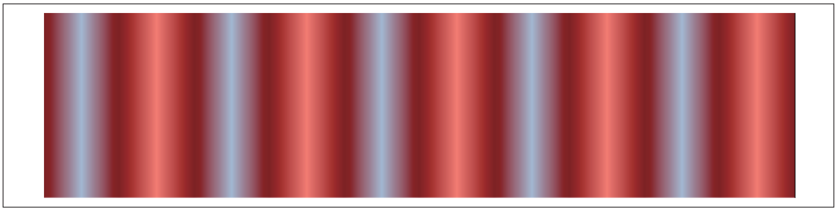


图 8-2：在两侧映射狭窄的渐变

尽管中间的渐变与图 8-1 中完全一样，但当渐变区域重复和映射时，在视觉上的效果会有不同，看起来就像一个颜色交替的金属管。需要注意的是，重复发生在你指定的渐变向量之前和之后两个方向上。

映射渐变的效果很大程度上会受到渐变矢量长度和色彩对比度级别的影响。图 8-2 创建了一个更精妙、波纹状的映射渐变，如图 8-3 所示。

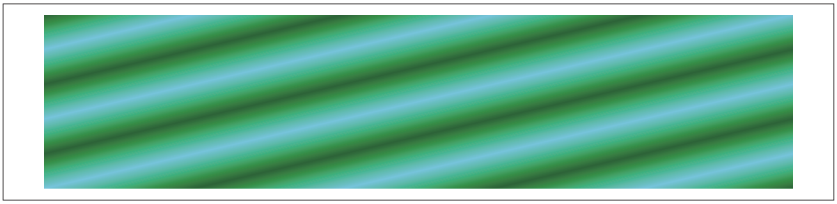


图 8-3：映射三种颜色的渐变

例 8-2　使用映射渐变来创建一个平滑重复的图案

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4in" height="1in">
    <title xml:lang="en">Reflecting Gradient</title>
    <linearGradient id="waves" spreadMethod="reflect"
                    x2="10%" y2="10%">
        <stop stop-color="darkGreen" offset="0"/>
        <stop stop-color="mediumSeaGreen" offset="50%"/>
```

```
        <stop stop-color="skyBlue" offset="100%"/>
    </linearGradient>
    <rect width="100%" height="100%" fill="url(#waves)" />
</svg>
```

渐变矢量沿着对角线从左上角（默认的 x1 和 y1 值）指向 10% 位置的点。这是单独一个渐变在单个周期的距离。因此所得到的形状有 5 对（共 10 次循环）交替的渐变，沿着对角线从绿到蓝交替。

8.3 非映射重复

映射渐变创建了平滑地从一种颜色变换为另一种颜色然后再变换回来的效果。相比之下，当 spreadMethod 的值为 repeat 时，渐变结点会从始至终以相同的顺序重复。除非开始和结束的颜色相同，否则会导致渐变看起来不连续。

我们依然采用例 8-1 中的代码，并使用 spreadMethod="repeat"，能够清晰地看到它们之间的差异。结果如图 8-4 所示。

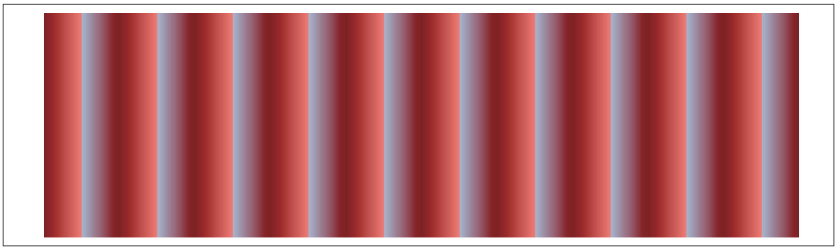


图 8-4：在两侧重复的狭窄渐变

一个明显的不同是现在每个渐变周期的边缘都创建了锐利的线条，就好像你看到的是手风琴的褶子，每侧的光线都不同。

重复渐变在创建条纹效果时非常有用。例 8-3 利用这一优点创建了一个条纹的背景效果，如图 8-5 所示。

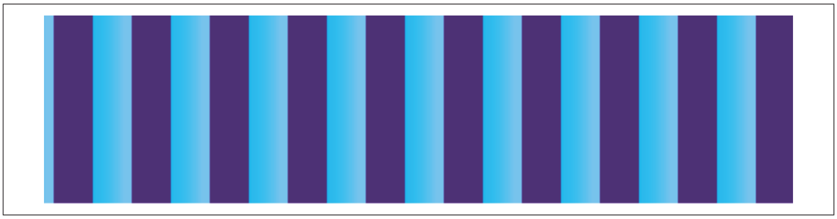


图 8-5：使用重复渐变创建的条纹背景

例 8-3　使用重复渐变创建一个条纹图案

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4in" height="1in">
    <title xml:lang="en">Repeating Gradient</title>
    <linearGradient id="wallpaper" spreadMethod="repeat"
                    gradientUnits="userSpaceOnUse"
                    x1="5px" x2="45px">                     ❶
        <stop stop-color="indigo" offset="50%"/>            ❷
        <stop stop-color="deepSkyBlue" offset="50%"/>
        <stop stop-color="lightSkyBlue" offset="90%"/>      ❸
    </linearGradient>
    <rect width="100%" height="100%" fill="url(#wallpaper)" />
</svg>
```

❶ 渐变使用用户空间单位来确定尺寸，所以每个渐变周期的宽度可以使用绝对单位来定义，而不是使用引用形状的百分比定义。渐变矢量是水平的（`y1` 和 `y2` 都使用默认的 0），每个周期的宽度是 40px，也就是与 `x1` 和 `x2` 之间的差值。

❷ 第一个结点在循环的中间位置（50% 偏移），每个循环周期起始点和偏移之间的空白会使用第一个结点的颜色填补。换句话说，每一个重复周期的前半部分都是使用 `indigo` 填充的纯色条纹。

❸ 在 50% 偏移的位置颜色急剧变为蓝色后，一直到 90% 偏移的位置浅蓝色有一个细微的渐变，循环周期的剩余部分将会使用浅蓝色填充。

例 8-3 还演示了我们在第 7 章中提到的一些内容：不设置渐变矢量的起始或终止位置的结点与改变矢量的长度是不同的（在重复或映射渐变中）。颜色是在 `<stop>` 和矢量终点之间填补的，而重复是应用在矢量本身上的。

CSS 与 SVG

重复 CSS 渐变

`repeating-linear-gradient()` 函数用于创建 CSS 中的重复线性渐变。参数与常规的线性渐变是一样的：一个方向参数（关键词或角度），接着是颜色结点列表。

> 重复 CSS 渐变在许多浏览器中的支持都晚于常规的渐变。但是现在所有主流浏览器的最新版本，包括 Safari（不支持重复 SVG 渐变）都已经支持。

重复部分完全匹配第一个和最后一个结点之间的距离。为了看到重复效果，

你需要明确设置至少一个结点的偏移，否则，单个周期将会填满整个 CSS 布局盒。

要想在重复之间创建纯色块，你需要提供两个颜色相同的结点。例如，要创建垂直的蓝色和靛蓝色条纹背景，代码如下：

```
background: repeating-linear-gradient(to right,
                indigo 5px, indigo 25px,
                deepSkyBlue 25px,
                lightSkyBlue 41px, lightSkyBlue 45px);
```

重复块依然是 40px 宽，即第一个和最后一个结点的差。中间的结点使用的也是 px 值。如果你使用的是百分比值，它们会相对于 CSS 布局盒而不是渐变重复距离来计算，完全不计缩放的话，50% 的偏移将会远大于 45px 的最终偏移。

CSS 中没有实现映射渐变的简单方式。要想创建映射图案，你需要手动复制 stop 来创建一个完整的重复块，代码如下：

```
background: repeating-linear-gradient(to bottom right,
                darkGreen, mediumSeaGreen 5%,
                skyBlue 10%,
                mediumSeaGreen 15%, darkGreen 20%);
```

本例中，第一个结点的偏移是默认的 0，因此重复图案是对角线长度的 20%，与例 8-2 中的往复循环一样。

对于 CSS 背景中的水平和垂直渐变，你还可以使用 background-size 来定义一个大于一次重复的背景图片，并使用 background-repeat 属性平铺填充元素，以此来创建重复渐变的效果。

相比于 repeating-linear-gradient，平铺的方式有两个便利之处。你可以独立于重复区域的尺寸，在渐变方向上使用百分比作为距离，并且对于重复区域起始和结束位置的纯色，你不需要使用额外的结点。例如，你可以使用如下简写的样式规则来创建背景效果：

```
background: linear-gradient(to right,
                indigo 50%,
                deepSkyBlue 50%,
                lightSkyBlue 90%)
                5px 0/40px 100% repeat-x;
```

如果你想要改变条纹的宽度或偏移，现在只需要调整一个值（40px 100% 的 background-size 的值用于设置总宽度，5px 0 的 background-position 的值用于设置偏移），而不用去修改每个结点。

基于一点点三角形学，你还可以通过这种方式重复对角渐变：仔细计算出正确的 background-size 来创建重复的磁贴（tile），但是舍入的效果可能会在磁贴之间创建可见的边。

8.4 在HTML中使用（复用）渐变

前面介绍的关于渐变的例子都是在 SVG 文件中使用渐变来填充简单的形状，而且大部分是简单地直接填充占据整个图形的长方形。然而，随着 CSS 渐变的广泛支持，使用简单的 SVG 渐变背景的情况不断减少。但同时，随着浏览器支持程度的提高，其他 SVG 图形的使用场景在不断增多。这包括内敛 SVG 代码，HTML 文件中通过主文档的样式表添加样式的标记。

渐变可以用在图标、图表或其他 HTML 页面内的 SVG 代码中。大多数情况下，它的运行结果与在单独的 SVG 文档中一样，但也有一些例外，大都可以归因于浏览器的 bug。

对于单个比较大的 SVG 文件（例如，数据可视化），所有的 SVG 代码，包括渐变的定义，通常都包含在一起。

相比之下，对于 SVG 图标系统，图标通常都被定义一次，然后通过 <user> 元素重复使用。这样可以保持你的页面结构，且使得主要的内敛标签清晰简洁。最简单情况下，每个图标实例可以通过仅比嵌入图片稍多的标签来表示：

```
<svg class="icon"><use xlink:href="#star" /></svg>
```

在同一文件的其他地方，添加 id="star" 的 <symbol> 标签将会定义实际的形状，它将会被缩放到适合添加 icon 类的 <svg> 元素的大小。

根据不同的使用情况，你可能需要引入一个标题提示、可访问名称以及备用的文本：

```
<svg class="icon" role="img" aria-labelledby="icon-0001">
    <use xlink:href="#star">
        <title>Tooltip</title>
        <desc id="icon-0001">Alternative Text</desc>
    </use>
</svg>
```

现代浏览器中 <title> 中的内容将作为工具提示。<desc> 中的内容用于可访问名称（因为有 aria-labelledby 属性）并且可以在旧的不支持 SVG 的浏览器中显示。

每个 `<symbol>` 中的原始图形通常只分配最少的样式规则，这样它们就可以继承 `<use>` 实例上设置的样式，包括一些交互样式，比如 `hover` 和 `focus` 效果。这些样式还包括引用渐变元素作为填充或描边。然而，正如之前在第 5 章中说的那样，URL 解析规则以及外部 SVG 资源的低支持也避免了这些样式在外部样式表中分配。

> 在除 Firefox 外的所有浏览器中，渲染服务与图形必须在同一个文档中，所以样式规则也必须定义在同一文档中，而不是在外部样式表中。

理论上，外部样式可以通过 `url()` 引用链接到主文档中。但在实践中，这与外部样式表是用于多个文档中重复使用的主要用途相矛盾。

由于它们不能集中在外部文件中，渐变本身通常通过它们自己 `<svg>` 元素内的不在屏幕上绘制任何东西的图标定义来收集。（它们必须定义在 `<svg>` 元素内才能被正确解析为 SVG 内容。）

> 正如我们介绍用户空间渐变时提到的，除 Firefox 外的所有浏览器都基于渐变元素的父级 `<svg>` 来缩放 `userSpaceOnUse` 渐变，而不是被绘制的形状的坐标系。因此用户空间渐变在使用一个单独的 SVG 定义时会失效。

为了获得最佳的浏览器支持且易于维护标记，SVG 通常放置在 HTML 中 `<body>` 的顶部。

> 旧的 WebKit 浏览器不能正确定位 `symbol` 和其他一些文档中定义在使用它的元素之后的可复用内容。

理想情况下，你可能会把 SVG 定义与其他不显示的内容一起放置在 HTML 的 `<head>` 中。至少，你也可以使用 `display:none` 来保证额外的 SVG 不会影响你的网页。但这两个方法在实践中都不可用。

> 所有被测试浏览器在渐变的祖先元素的 `display` 设置为 `none` 时都不会渲染渐变（或图案），尽管 SVG 规范中明确声明 `display` 不应该有任何效果。这是 SVG 在跨浏览器一致性上最令人沮丧的一个例子。

所以，你需要使用替代的 CSS 来保证 SVG 定义不会影响你的网页，并使用 `ARIA` 属性来确保辅助的技术不会处理它。此外，为了避免 CSS 加载时间过长，你可能还需要使用 SVG 属性来摧毁定义元素的大小。最后，由于 IE 实现了未敲定的 SVG 1.2 中对于键盘控制 SVG 的提议，你还需要使用 `focusable` 属性来明确告诉它 SVG 不应该成为键盘的焦点。

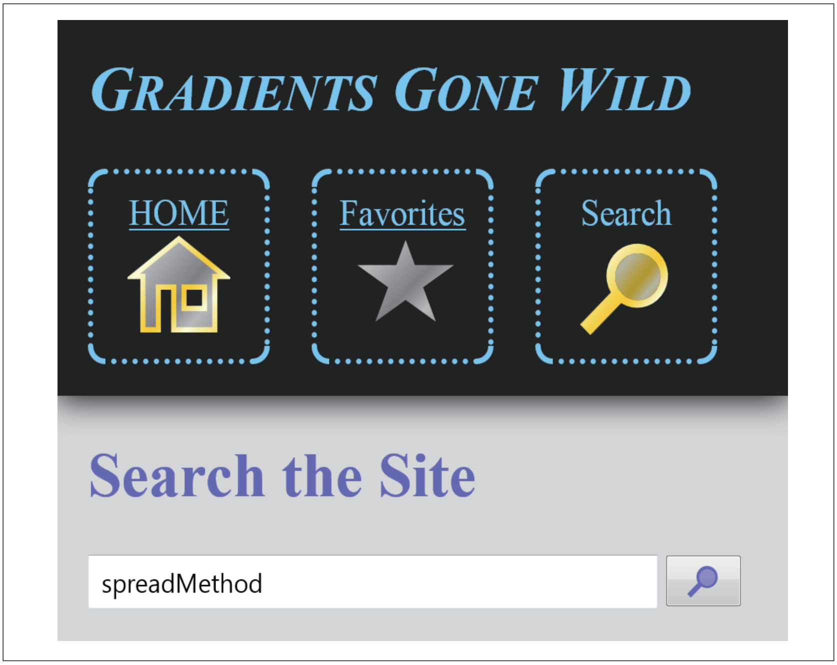


图 8-6：HTML 页面中使用渐变来填充和描边的图标

> 在 IE 中，HTML 中所有的 `<svg>` 元素默认都是键盘聚焦的，只能通过 `focusable="false"` 来从 `tab` 的索引中删除。在最新版本的 WebKit/Blink 浏览器中，`<svg>` 元素默认是键盘不能聚焦的，但是可以通过给 `tabindex` 设置一个正的值来使它可以获得焦点。Firefox 没有实现任何一种焦点控制的方式。

有了这么多警示，那么 HTML 文件中给 SVG 图标使用渐变的结果是什么样的呢？例 8-4 给出了在导航菜单中使用 SVG 图标的网页的核心标记和样式。不同颜色的渐变用于区分导航选项中的当前站点，以及指示出悬停和获得

焦点时的状态。图 8-6 显示了在第一个导航链接处于焦点状态的样子。

例 8-4　HTML 页面内使用渐变的 SVG 图标

HTML 标记：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>Re-using Symbols with Gradients</title>
    <style type='text/css'>
        /* styles must be included in the same document */
    </style>
</head>
<body>
    <svg class="defs-only" aria-hidden="true" focusable="false"
         width="0" height="0">                                      ❶
        <linearGradient id="silver-shine" spreadMethod="repeat"
                        gradientTransform="rotate(40) scale(0.8)" >
            <stop offset="0" stop-color="gray" />
            <stop offset="0.35" stop-color="silver" />
            <stop offset="1" stop-color="gray" />
        </linearGradient>
        <linearGradient id="gold-shine"
                        xlink:href="#silver-shine" >                ❷
            <stop offset="0" stop-color="gold" />
            <stop offset="0.35" stop-color="lightYellow" />
            <stop offset="1" stop-color="gold" />
        </linearGradient>
        <symbol id="home" viewBox="0 0 200 200">                    ❸
            <path d="M30,180 V80 H10 L100,10 190,80 H170 V180
                   H90 V100 H60 V180 H30 Z
                   M110,100 H150 V140 H110 Z"
                 fill-rule="evenodd" />
        </symbol>
        <symbol id="star" viewBox="10 10 170 150">
            <path d="M100,10 L150,140 20,50 180,50 50,140 Z" />
        </symbol>
        <symbol id="magnify" viewBox="0 0 200 200">
            <path d="M10,170 L70,110 A60,60 0 1,1 90,130 L30,190Z
                   M85,104 A44,44 0 1,1 96,116 Z"
                 fill-rule="evenodd" />
            <path d="M85,104 A44,44 0 1,1 96,116 Z" fill-opacity="0.7" />
        </symbol>
    </svg>
    <a class="skip-nav" href="#main">Skip to Main Content</a>
    <header>
        <h1>Gradients Gone Wild</h1>
```

```
        <nav>
            <ul>
                <li><a href="/">HOME
                        <svg class="nav-icon" role="presentation">
                            <use xlink:href="#home" />              ❹
                        </svg>
                    </a>
                </li>
                <li><a href="/favorites">Favorites
                        <svg class="nav-icon" role="presentation">
                            <use xlink:href="#star" />
                        </svg>
                    </a>
                </li>
                <li><a title="(You are here)">Search              ❺
                        <svg class="nav-icon" role="presentation">
                            <use xlink:href="#magnify" />
                        </svg>
                    </a>
                </li>
            </ul>
        </nav>
    </header>
    <main id="main">
        <h1>Search the Site</h1>
        <form>
            <input type="search" name="q"
                   aria-label="Enter Search Terms"/>
            <button type="submit" aria-label="Search">
                <svg>
                    <use xlink:href="#magnify" />                  ❻
                    GO
                </svg>
            </button>
        </form>
    </main>
</body>
</html>
```

❶ 初始的 `<svg>` 用于包含复用内容的定义。除了会触发 CSS 隐藏的 `defs-only` 类外，还添加了 `aria-hidden="true"` 属性，并设置标记的宽和高都为 0。`focusable="false"` 使 IE 浏览器不能通过键盘使它获得焦点。

❷ 两个渐变几何上完全一样，`spreadMethod` 和 `gradientTransform` 属性在 `xlink:href` 引用的时候将会复制到另一个上。但是它们的颜色不同，第二个渐变中 `<stop>` 元素会完全替换第一个中对应的结点。

❸ 三个 <symbol> 元素用于定义图标，每个都有 viewBox 属性，所以它们将会被缩放到适合背景的大小。

❹ 在主要的网页标记中，每个导航图标都把 <svg> 嵌套在导航列表中的 <a> 元素内。单独的 <use> 元素用于引用 symbol，由于 <user> 元素上没有位置属性，所以复用的 symbol 会被拉伸到正好适合 <svg> 的大小。

❺ 当前页面表示的是导航列表中没有 href 属性的锚元素，因此它是一个不合法的超链接。

❻ 按钮图标的标记用法与导航中类似，外观上有差异是因为 SVG 继承样式和缩放不同。在现代浏览器中，按钮中没有可见的文字，aria-label 属性用于给它添加可访问名称。SVG 中的纯文本内容（GO）为旧版本浏览器提供了一种降级方案。

相关的 CSS 样式：

```
svg.defs-only {
    display: block;
    position: absolute;
    height: 0; width: 0;
    overflow: hidden;                    ❶
}
svg.nav-icon {
    display: block;
    width: 3em;
    height: 3em;
    margin: auto;
    fill: url(#silver-shine);            ❷
}
a:not(:link) > .nav-icon {
    fill: url(#gold-shine);              ❸
}
nav a:link:focus, nav a:link:hover {
    stroke: url(#gold-shine);
    stroke-width: 10px;                  ❹
}
input, button {
    display: inline-block;
    height: 2em;
    padding: 0 0.5em;
}
input[type="search"] {
    width: calc(100% - 5em);
}
button {
```

```
        display: inline-block;
        width: 3em;
        vertical-align: top;
        color: inherit;
    }
    button svg {
        height: 100%;
        width: 100%;
        fill: currentColor;                ❺
    }
```

❶ defs-only 类补充了一些标记的属性，来确保仅用于定义的 <svg> 元素无论如何都不会影响布局或视觉效果。

❷ 用于导航图标的 <svg> 元素拥有固定的宽和高。fill 引用了 SVG 中预先设置的一个渐变，它将会被 <use> 元素以及复用的 symbol 继承。

❸ :link 伪类用于区分 <a> 元素是否有合法超链接。如果类没有匹配，SVG 内会重新使用更明亮一些的金色渐变来突出这是一个活跃状态的页面。

❹ 对于其他图标，当链接在鼠标光标悬停或获得焦点时将会添加一个金色的描边来表示交互的效果。为了证实这种效果，样式直接设置在了 <a> 元素上。它们同样会被 SVG 及其内容继承，因为所有 SVG 元素都没有设置其他的描边样式。10px 的描边宽度将会应用到最终添加描边的图形的坐标系中，这种情况下，坐标系是定义在每个 symbol 内的。

❺ 因为样式在使用时都会被继承，所以当图形应用在同一个页面的不同背景中时是可以改变的。用在表单按钮内的搜索图标将会使用从 <button> 上继承来的文本的颜色来设置样式。

因为 SVG 标记的定义和样式在网站上的每页内都相同，所以我们通常在编译时会进行一些服务端的处理。如果 SVG 代码比较多，或大多数用户在网站上访问多个网页，我们可以在客户端使用 Ajax 脚本，使 SVG 文件可以被缓存到浏览器中。

> 许多流行的 SVG 图标系统仅在 <use> 元素引用外部文件不被支持时，使用 Ajax 来引入 SVGsymbol。WebKit 和 Blink 浏览器目前支持外部 symbol 的引入，但是不支持渐变以及其他渲染服务的引入。如果有必要，一定要确保你的脚本中有导入渐变和图案的具体规则。

主要图形的定义都包含在一个单独的标记块内，这样很容易给每个图标生

成新的实例，或给这些实例使用不同的渐变。

例 8-5 使用与例 8-4 相同的渐变，以及其中一个完全相同的图标，依据 HTML 代码中的数据属性来动态生成星级图。渐变和 `symbol` 的定义是静态标记，但是 `<use>` 实例是基于数据生成的。

因为现在每个 SVG 内都包含了每个图标的多个副本，所以 `<svg>` 上的 `viewBox` 以及 `<use>` 元素上的位置属性都是正确缩放内容所必需的。此外，`ARIA` 属性用于把评级信息传递给辅助技术，用只读的范围滑块作为标记。每个等级（图形中高亮星星的个数）的视觉效果直接由 `aria-valuenow` 属性决定，使用的是基于 `nth-of-type` 选择器的 CSS 规则。图 8-7 显示的是最终的网页。



图 8-7：动态生成基于渐变效果的 SVG 星级图标

例 8-5　使用动态插入 SVG 图标的渐变

HTML 标记：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
```

```
        <title>Reusing Symbols with Gradients</title>
        <style type='text/css'>
            /* styles must be in the same document */
        </style>
    </head>
    <body>
        <svg class="defs-only" aria-hidden="true" focusable="false"
             width="0" height="0">                                     ❶
            <linearGradient id="silver-shine" spreadMethod="repeat"
                            gradientTransform="rotate(40) scale(0.8)" >
                <stop offset="0" stop-color="gray" />
                <stop offset="0.35" stop-color="silver" />
                <stop offset="1" stop-color="gray" />
            </linearGradient>
            <linearGradient id="gold-shine" xlink:href="#silver-shine">
                <stop offset="0" stop-color="gold" />
                <stop offset="0.35" stop-color="lightYellow" />
                <stop offset="1" stop-color="gold" />
            </linearGradient>
            <symbol id="star" viewBox="0 0 200 200">
                <path d="M100,10 L150,140 20,50 180,50 50,140 Z" />
            </symbol>
        </svg>
        <h1>Image Format Awesomeness:</h1>
        <ol>
            <li id="svg" data-rating="5">                              ❷
                <abbr title="Scalable Vector Graphics">SVG</abbr>
            </li>
            <li id="png" data-rating="4">
                <abbr title="Portable Network Graphics">PNG</abbr>
            </li>
            <li id="gif" data-rating="3">
                <abbr title="Graphics Interchange Format">GIF</abbr>
            </li>
            <li id="bmp" data-rating="1">
                <abbr title="Bitmap (Windows)">BMP</abbr>
            </li>
        </ol>
        <script>
            /* script could be loaded as a separate file */
        </script>
    </body>
    </html>
```

❶ 渐变的定义与例 8-4 中相同，但这里我们只需要一个 `<symbol>`。

❷ 网页主体的标记中没有初始化任何 SVG 的内容，而列表的每一项通过 `data-rating` 属性来区分。

CSS 样式：

```
html {
    background-color: #222;
    color: lightSkyBlue;
}
svg.defs-only {
    display: block;
    position: absolute;
    height: 0; width: 0;
    overflow: hidden;                    ❶
}
ol{
    padding: 0;
}
li{
    display: block;
    text-align: center;
    list-style: none;
    font-size: larger;
}
svg.star-rating {
    display: block;
    margin: auto;
    width: 10em;
    max-width: 100%;
    height: auto;
    max-height: 2em;                     ❷
}
.star {
    fill: url(#silver-shine);            ❸
}
[aria-valuenow="1"] .star:nth-of-type(-n+1),
[aria-valuenow="2"] .star:nth-of-type(-n+2),
[aria-valuenow="3"] .star:nth-of-type(-n+3),
[aria-valuenow="4"] .star:nth-of-type(-n+4),
[aria-valuenow="5"] .star:nth-of-type(-n+5)
{
    fill: url(#gold-shine);              ❹
}
[data-rating]:not(.initialized)::after {
    display: block;
    color: darkgoldenrod;
    text-shadow: black 1px 1px 1px,
                 gold 0 0 3px;
    content: attr(data-rating);          ❺
}
```

❶ 与例 8-4 中相同的 svg.defs-only 样式用于隐藏包含所有图形定义的元素。

❷ 给要包含星级的 SVG 元素设定固定的宽，如果有必要的话还可能被缩放；max-height 确保在不支持基于 SVGviewBox 自动缩放的浏览器中也有一个合适的高度。

❸ 星星图标默认使用 silver-shine 渐变。

❹ 星星的正确数量变为金色，用一系列选择器与 aria-valuenow 属性绑定。例如，:nth-of-type(-n+3) 选择组内前三个 <use> 图标，它们是 n 为正整数时根据索引匹配到的图标元素。

❺ 如果脚本没有执行，即 SVG 星星没有生成，CSS 伪元素将会把 data-rating 属性的值输出到屏幕上。

JavaScript：

```
(function(){
    var ns = {svg:"http://www.w3.org/2000/svg",
              xlink:"http://www.w3.org/1999/xlink"};
    var maxRating = 5;
    var index = 0;

    var ratings = document.querySelectorAll("[data-rating]");  ❶
    for (var i=0, n=ratings.length; i<n; i++){
        var r = ratings[i];
        //add an `id` value if it doesn't exist
        r.id = r.id || "rating-" + (index++);

        //parse the rating
        var value = parseInt(r.getAttribute("data-rating"),10);

        //create and insert an SVG to represent the rating
        var s = document.createElementNS(ns.svg, "svg");       ❷
        s.setAttribute("viewBox", "0 0 " + maxRating + " 1");
        s.setAttribute("class", "star-rating");
        s.setAttribute("role", "slider");
        s.setAttribute("aria-labelledby", r.id);
        s.setAttribute("aria-valuemin", 0);
        s.setAttribute("aria-valuemax", maxRating);
        s.setAttribute("aria-valuenow", value);
        s.setAttribute("aria-readonly", true);

        //create a group and give it a tooltip title
        var g = document.createElementNS(ns.svg, "g");
        s.insertBefore(g, null);
        var t = document.createElementNS(ns.svg, "title");
```

```
        t.textContent = value + " out of " + maxRating;         ❸
        g.insertBefore(t, g.firstChild);

        //create and insert the stars into the group
        for (var j=0; j<maxRating; j++) {
            var u = document.createElementNS(ns.svg, "use");
            u.setAttribute("class", "star");
            u.setAttributeNS(ns.xlink, "href", "#star");
            u.setAttribute("width", "1");
            u.setAttribute("x", j);
            g.insertBefore(u, null);                            ❹
         }

        r.insertBefore(s, null);
        r.classList.add("initialized");                         ❺
    }
})();
```

❶ 该脚本选中所有拥有 `data-rating` 属性的元素，然后循环它们。每一个都在没有 id 时任意分配一个 id，这在 ARIA 引用时有用。

❷ SVG 元素在创建时都设置了 `viewBox`、`class`、`ARIA` 等属性，同时还把从数据属性上分析出的评级值以及最大评级值作为常数存储在脚本中。

❸ `<title>` 元素用于把评级翻译为工具提示文本。因为许多浏览器在 `<title>` 直接作为 `<svg>` 的子元素时不把它显示为提示工具，额外的组（`<g>`）用于包裹 `title` 和星星图标。

❹ 星星图标本身（`<use>` 元素）都是一样的，除了沿着 SVG 的宽度把它们分隔开的 `x` 属性，图标都被添加到 `<g>` 内。

❺ 整个图形都被插入到拥有 `data-rating` 属性的元素的最后。然后该元素通过 `initialized` 类来避开降级的 CSS 样式。

每个 SVG 动态生成的内容都一样，除了 `aria-valuenow` 和 `aria-labelledby` 属性以及 `<title>` 的内容。写为标记的话，如下所示：

```
<svg viewBox="0 0 5 1"
     role="slider" aria-labelledby="png"
     aria-valuemin="0" aria-valuemax="5"
     aria-valuenow="4" aria-readonly="true">
    <g>
        <title>4 out of 5</title>
        <use class="star" xlink:href="#star" width="1"/>
        <use class="star" xlink:href="#star" width="1" x="1"/>
        <use class="star" xlink:href="#star" width="1" x="2"/>
        <use class="star" xlink:href="#star" width="1" x="3"/>
```

```
        <use class="star" xlink:href="#star" width="1" x="4"/>
    </g>
</svg>
```

这种结构很适合于使用标签模板来生成动态内容的 JavaScript 库。但是，一定要确定你使用的模板工具能够恰当地识别 SVG 元素并且给它们分配 SVG 命名空间；否则，可能你最终生成的动态 DOM 拥有正确的标签名和属性，但是没有生成 SVG 图像。

第 9 章

径向渐变

之前介绍过，线性渐变是依据一条线（渐变矢量）的坐标来定义的。颜色结点沿着这条线来设定位置。每种颜色的值在直线的两侧都会无限延伸。通过调整渐变结点和操作矢量，线性渐变可以创建许多效果。但 SVG 中的渐变不只这一种。

径向渐变也是一种渐变效果，在这里，颜色改变沿着一点向外辐射。这些渐变可以创建发光的灯的效果，或者可用于表示球体和类似圆形结构的阴影。

径向渐变最简单的一种形式是通过定义一个圆，然后颜色从圆心到边缘逐渐改变。这也是 SVG 中 `<radialGradient>` 元素的默认行为。通常，默认行为并不是唯一的选择，你可以通过使用 SVG 径向渐变创建多种效果，包括一些目前还不能通过 CSS 渐变来实现的效果。

本章介绍所有可能实现的效果，略过了径向渐变与线性渐变的相似部分而专注于它们之间的区别。最后一节的大型图片，向你展示了如何结合多种不同渐变来创建复杂的场景。

9.1 径向渐变基础

径向渐变与线性渐变在结构上类似，至少在标签上很相似。与 `<linearGradient>` 一样，`<radialGradient>` 是 `<stop>` 元素的容器，每个 `<stop>` 都有一个介于 0 到 1（0% 到 100%）之间的 `offset` 属性。结点的值同样通过 `stop-color` 和 `stop-opacity` 样式或表现属性来指定。

它们的不同之处在于这些结点的偏移映射到要填充的二维空间的方式。

对于径向渐变，偏移是起始点到结束圆的距离的一部分。你可以修改起始点（称为焦点）的位置以及结束圆的大小和位置。

默认情况下，当 <radialGradient> 没有任何位置属性时，结束圆是填充对象边界盒的最大圆，焦点就是它的中心。

例 9-1 改编了 4 个在第 6 章介绍结点和偏移时用到过的基本的红蓝渐变；结点没有变，只不过把它们放到了 <radialGradient> 元素里面。

> 如果例 9-1 中的径向渐变和对应的线性渐变在同一个文件中，我们可以使用 xlink:href 来复制结点，即使它们是不同的渐变类型。

例 9-1 中的渐变用于填充圆，结果如图 9-1 所示。

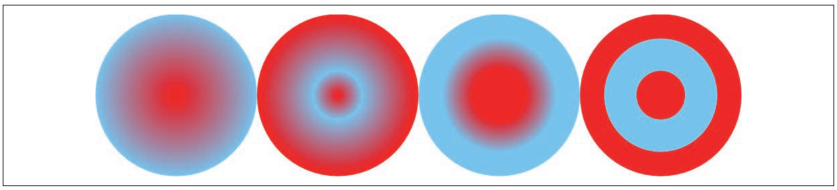


图 9-1：各种结点图案的径向渐变

例 9-1　创建径向渐变来填充圆

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4in" height="1in">
    <title xml:lang="en">Radial Gradients</title>
    <radialGradient id="red-blue">
        <stop stop-color="red" offset="0"/>
        <stop stop-color="lightSkyBlue" offset="1"/>
    </radialGradient>
    <radialGradient id="red-blue-2">
        <stop stop-color="red" offset="0"/>
        <stop stop-color="lightSkyBlue" offset="0.3"/>
        <stop stop-color="red" offset="1"/>
    </radialGradient>
    <radialGradient id="red-blue-3">
        <stop stop-color="red" offset="0.3"/>
        <stop stop-color="lightSkyBlue" offset="0.7"/>
    </radialGradient>
    <radialGradient id="red-blue-4">
        <stop stop-color="red" offset="0.3"/>
```

```
            <stop stop-color="lightSkyBlue" offset="0.3"/>
            <stop stop-color="lightSkyBlue" offset="0.7"/>
            <stop stop-color="red" offset="0.7"/>
        </radialGradient>
        <circle r="0.5in" cy="50%" cx="12.5%" fill="url(#red-blue)" />
        <circle r="0.5in" cy="50%" cx="37.5%" fill="url(#red-blue-2)" />
        <circle r="0.5in" cy="50%" cx="62.5%" fill="url(#red-blue-3)" />
        <circle r="0.5in" cy="50%" cx="87.5%" fill="url(#red-blue-4)" />
    </svg>
```

颜色结点没有绘制为平行线，而是被绘制为同心圆。当颜色过渡比较锐利时，就像最后一个例子那样，将创建出靶心的效果。

9.2 填充盒子

当使用径向渐变来填充圆时，渐变的结束圆（即将要填充对象边界盒的最大圆）就是圆本身。因此渐变可以完美适应形状。当情况不是这样时，额外的部分是根据 spreadMethod 属性来填充的，默认情况下，会使用最后一个结点颜色来填补。

图 9-2 显示了用例 9-1 中的渐变填充 1 平方英寸的长方形。

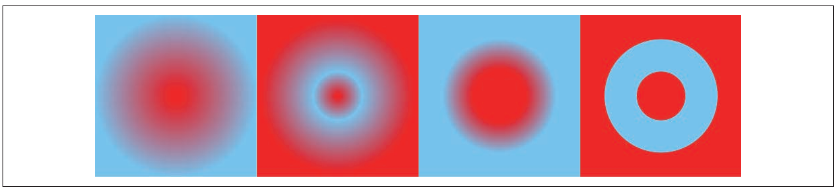


图 9-2：使用径向渐变填充正方形

径向渐变也可以很好地填充 <ellipse> 元素，如图 9-3 所示，现在每个结点都代表一个同心椭圆。

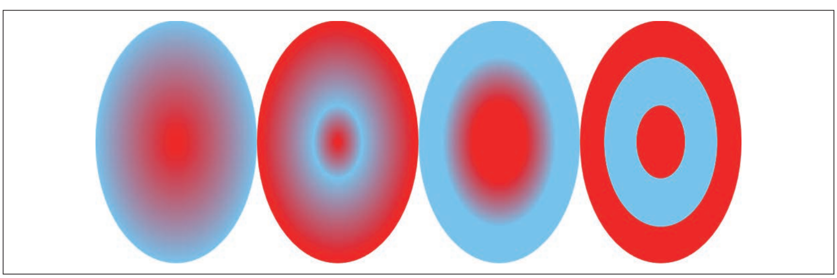


图 9-3：使用径向渐变填充椭圆

我们同样可以看到使用渐变填充椭圆边界盒大小的长方形时的椭圆图案，如图 9-4 所示。

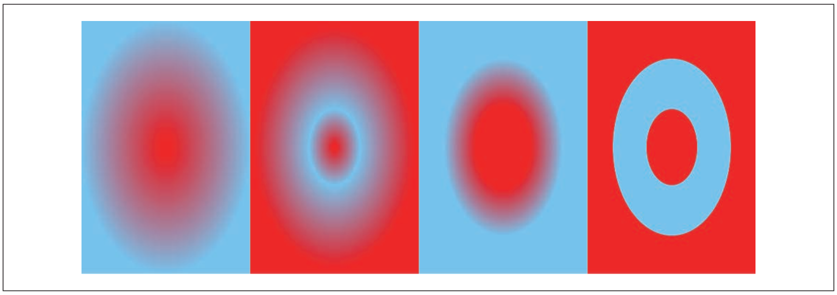


图 9-4：使用径向渐变填充长方形（非正方形）

图 9-3 和图 9-4 的结果使用都是例 9-1 中定义的渐变，在渐变元素上没有属性，表明应该使用椭圆图案来填充。相反，椭圆图案是拉伸对象边界盒单位创建的坐标系的结果。结点图案依然是数学上的“圆”，结点的每一点到中心点的距离都相同，不过当你度量它时，它已经在拉伸后的坐标系中了。

创建总是沿着完美圆的径向 SVG 渐变而忽略边界盒尺的唯一方式，是使用 `gradientUnits="userSpaceOnUse"`。然后，你需要使用其他属性来改变渐变的大小和位置来匹配它填充的形状。

当填充正方形或长方形时，100% 的结点偏移的位置是沿着刚刚触碰到每一条边的圆（有可能被拉伸）。但在图片中，纯色内边距隐藏了这个边界。`spreadMethod` 属性同样可以用来改变内边距效果，且另外的可选值也是 `repeat` 和 `reflect`。

正如第 8 章中提到的那样，编写本书时，WebKit 浏览器不支持重复和镜像渐变。旧版本的其他浏览器（2014 年之前的 Firefox 和 2013 年之前的 Chrome）将忽略 `spreadMethod` 属性的值而直接填补渐变。

当重复或映射径向渐变时，你不会以重复圆的虚线图案结束。相反，颜色图案会沿着每条射线（从中心点向外辐射）重复或镜像地扩展，直到到达形状的边缘。当径向渐变用于填充长方形时，它会动态改变拐角的样子。例 9-2 中使用了不同的选项，效果如图 9-5 所示。

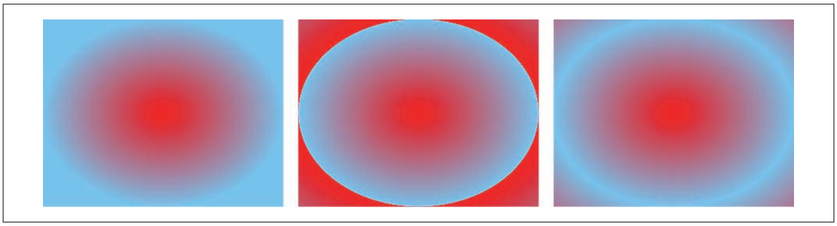


图 9-5：使用不同扩展方式的径向渐变：填补（左），重复（中），镜像（右）

例 9-2　使用 spreadMethod 改变拐角效果

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="1in">
    <title xml:lang="en">Repeating Radial Gradients</title>
    <radialGradient id="red-blue" spreadMethod="pad">
        <stop stop-color="red" offset="0"/>
        <stop stop-color="lightSkyBlue" offset="1"/>
    </radialGradient>
    <radialGradient id="repeat" spreadMethod="repeat"
                    xlink:href="#red-blue" />
    <radialGradient id="reflect" spreadMethod="reflect"
                    xlink:href="#red-blue" />

    <rect height="1in" width="32%" x="0" fill="url(#red-blue)" />
    <rect height="1in" width="32%" x="34%" fill="url(#repeat)" />
    <rect height="1in" width="32%" x="68%" fill="url(#reflect)" />
</svg>
```

在重复渐变中，颜色沿着圆 / 椭圆的边缘瞬间切换，重新以第一个颜色结点开始。在映射渐变中，边缘在颜色变回去之前到达顶峰。要真正看到这些效果，我们首先需要创建一个渐变圆没有填满整个边界盒的径向渐变。

CSS 与 SVG
CSS 中的径向渐变

CSS 同样也有径向渐变，使用 radial-gradient 函数来定义。与线性渐变一样，它的语法和值与 SVG 渐变相似但不同。

CSS 径向渐变的一个重要属性（目前 SVG 中不支持）是可以选择圆形或椭圆形的渐变形状。radial-gradient 函数的第一个参数用于描述最终的形状。它包括关键词 circle 来要求渐变必须沿着完美的圆，而忽略生成图片的宽高比，还有关键词 ellipse 来依据盒子拉伸渐变。

非正方形盒子内的完美的圆不会触及到盒子的所有边。因此形状参数还包括尺寸信息，它们可用于圆和椭圆。定义尺寸最简单的方式是使用以下四个尺寸关键词之一：closest-side，farthest-side，closest-corner，farthest-corner。

以逗号来结束形状和渐变的尺寸等信息，紧接着是一个逗号分隔的颜色结点列表。例如，如下的背景样式规则生成图 9-6 所示的渐变。

```
background: radial-gradient(circle closest-side,
                            red, lightSkyBlue);
```

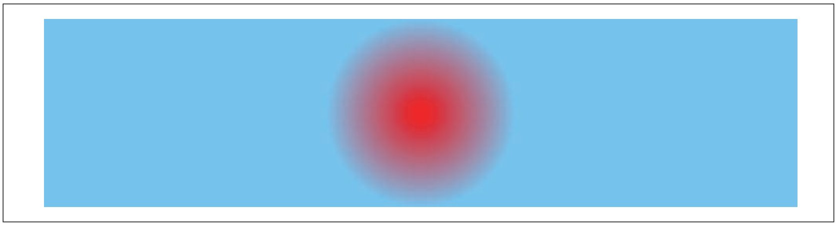


图 9-6：圆形 CSS 渐变

默认的形状是 ellipse，而默认的尺寸是 farthest-corner，所以如下的渐变都是相等的。

```
background: radial-gradient(red, lightSkyBlue);
background: radial-gradient(ellipse,
                            red 0%, lightSkyBlue);
background: radial-gradient(farthest-corner,
                            red, lightSkyBlue 100%);
background: radial-gradient(farthest-corner ellipse,
                            red 0%, lightSkyBlue 100%);
```

使用 farthest-corner 意味着没有空的拐角要填补，也意味着最后一个颜色结点只可以在拐角可见，如图 9-7 所示。

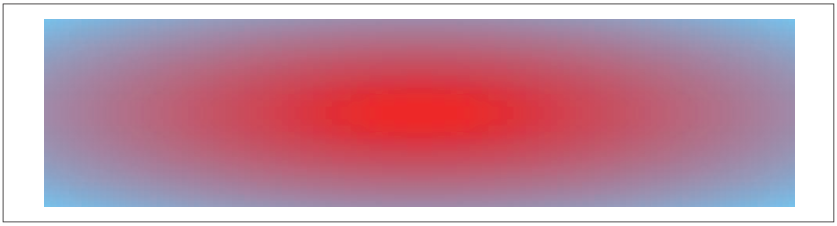


图 9-7：椭圆 CSS 渐变

个别颜色结点的位置可以使用有单位的长度或百分比来定义。当使用百分比时，它们是圆 / 椭圆半径的百分比。当在椭圆渐变中使用长度时，它们是沿着椭圆水平半径度量的。如果没有定义偏移，颜色在可用的空间内均匀分布。

与线性渐变类似，CSS 还有一个单独的 `repeating-radial-gradient` 函数。它的参数与正常的径向渐变相同，渐变每个周期的尺寸可以通过第一个和最后一个偏移的差来计算。

> 当第一个偏移非零时，Blink 和 WebKit 浏览器目前将使用纯色填充圆的中心。Firefox 和 IE 会向内向外重复渐变，依照声明中的通用指令（对于所有类型的渐变）来在两个方向上重复周期。

再次说明，要想创建映射渐变，你必须使渐变圆变为两倍大，然后手动添加两次颜色结点。

9.3 缩放圆

现在，你知道了线性渐变的默认位置和大小可以通过渐变元素上的属性来修改。对于径向渐变也同样如此。就像 `<linearGradient>` 的位置属性看起来很像 `<line>` 的属性那样，`<radialGradient>` 的位置属性看起来和 `<circle>` 很像。

`<radialGradient>` 上的 `cx`、`cy` 以及 `r` 属性定义了 100% 偏移的圆的大小和位置。它们的默认值都是 50%，所以会创建一个以坐标系中心为圆心且填充其整个宽度和长度的圆。默认情况下，焦点（0% 偏移颜色的点）会根据 `(cx,cy)` 点移动。

例 9-3 修改了这些属性创建了一个小的圆形渐变，且偏离 `userSpaceOnUse` 坐标系的中心。这在渐变圆的边缘和形状的边缘之间留下足够的空间，`reflect` 扩展方式可以创建一个非常好的波纹效果，就像雨滴落在积水上的波浪，如图 9-8 所示。

例 9-3　控制径向渐变的尺寸和位置

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4in" height="2in">
    <title xml:lang="en">Repeating Radial Ripples</title>
    <radialGradient id="raindrop" spreadMethod="reflect"
```

```
                        gradientUnits="userSpaceOnUse"
                        cx="30%" cy="70%" r="15px">
            <stop stop-color="lightSteelBlue" offset="0.4"/>
            <stop stop-color="darkSlateGray" offset="0.8"/>
            <stop stop-color="darkSlateBlue" offset="1"/>
        </radialGradient>
        <rect height="100%" width="100%" fill="url(#raindrop)" />
    </svg>
```

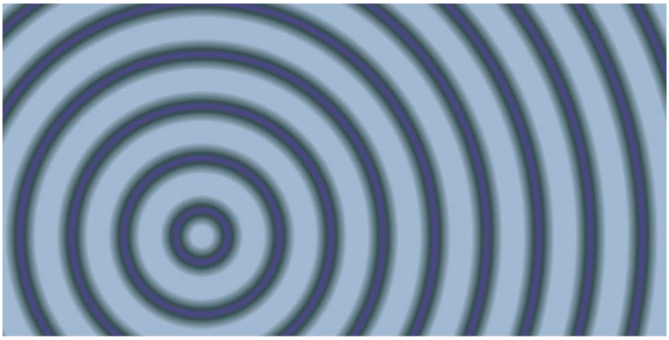


图 9-8：固定圆的大小的映射径向渐变

基本圆的半径是 15px，所以每隔 15px 的距离，渐变循环一次结点列表——正序或者倒序。图中暗色的环以 30px 分隔开，该距离是映射渐变在两个方向上各循环一个周期所需的距离。

相比而言，使用 `spreadMethod="repeat"` 可以更清晰地看到单个重复（15px 间隔），如图 9-9 所示。

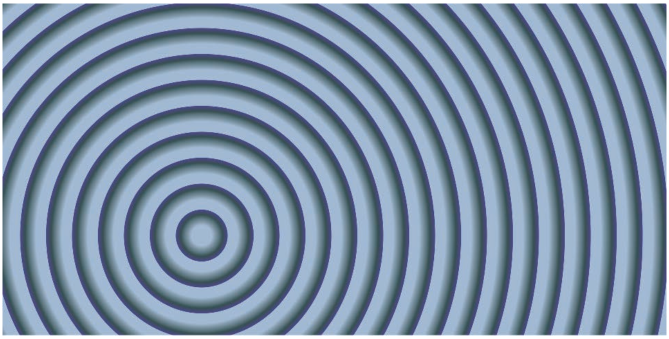


图 9-9：固定圆的大小的重复镜像渐变

CSS 与 SVG
控制 CSS 径向渐变的尺寸和位置

你可以通过给 `radial-gradient` 或 `repeating-radial-gradient` 函数的第一个参数添加更多信息，来控制 CSS 渐变圆形的尺寸和位置。

第一个参数完整的语法如下所示：

```
[ <ending-shape> || <size> ]? [ at <position> ]?
```

翻译为汉语的意思是：形状和尺寸是可选的，且顺序也没有严格规定，之后可选的是关键词 `at` 以及位置。

形状是关键词 `circle` 和 `ellipse` 其中之一。尺寸可以是一个 `closest/farthest-side/corner` 关键词或确切的半径长度。圆形有一个半径长度，而椭圆有两个：第一个是水平半径，然后是垂直半径。在给尺寸设置长度时，形状关键词不是必需的，因为它是由长度值来决定的。

对于椭圆渐变，两个半径值还可以分别被指定为图片宽和高的百分比。目前你无法使用百分比来指定圆形 CSS 渐变的尺寸。未来的 CSS 渐变可能会采用 SVG 相对于对角线长度来计算圆形半径百分比的方法，现在这可以用在 CSS 形状模块的圆中。

CSS 径向渐变的位置相当于 SVG 中的 `cx` 和 `cy` 属性。它可能是长度值或百分比，也可能是指定 CSS 背景位置的关键词的组合。关键词 `at` 必须要有，用于区分位置信息和尺寸信息。默认的位置是 `at center`，或 `at 50% 50%`。

> 当把渐变定位在边上或拐角（例如，在右上角）时，到 `closest-side` 或 `closest-corner` 的距离可能是零，这会把圆形渐变折叠为一个点，椭圆渐变折叠为线性渐变。

基于以上信息（注意 CSS 没有映射渐变，只有简单的重复渐变），例 9-3 中积水上的波纹渐变可以使用如下代码创建：

```css
background: repeating-radial-gradient(
            30px at 30% 70%,
            lightSteelBlue 0%, lightSteelBlue 20%,
            darkSlateGray 40%, darkSlateBlue 50%,
            darkSlateGray 60%,
            lightSteelBlue 80%, lightSteelBlue 100%);
```

圆的半径变为两倍（从 15px 变到 30px）以留下映射渐变的空间。类似地，偏移百分比减半（SVG 渐变中的 100% 变为 CSS 渐变圆形的 50%），复制一份结点来填充圆的剩余部分。不同于重复线性渐变，你不必担心百分比偏移打乱渐变的比例，它们一直都是你指定的形状半径的百分比。

9.4　调整焦点

对于径向渐变还有另外两个位置属性：`fx` 和 `fy`。他们是渐变焦点的坐标。如果说渐变圆是描述 100% 偏移处结点的位置，那么焦点描述的就是 0% 处结点的位置。

我们之前看到的渐变使用的都是默认的焦点，通常是圆心。`fx` 的默认值是 `cx` 的值，`fy` 的默认值是 `cy` 的值。任何其他值都会创建一个不对称的渐变。

焦点不在圆心意味着什么呢？它意味着渐变圆的长度在每个方向上都是不同的。从该点出发的每条射线在到达圆的边缘时会通过所有的结点颜色。

> 焦点应该在渐变圆内，如果不这样，根据 SVG 1.1，它将被移动到与它最近的圆的边缘上。SVG 2 基于目前在 HTML canvas 渐变函数中使用的方式提出了不同的方案。换句话说，如果焦点在圆的外部，就不要指望跨浏览器有一致的效果了。

焦点不在圆心看起来是什么样子呢？效果渐变是在一个方向上被压缩了而在另一个方向上被放大。这可能是你能看到的最简单的效果。

例 9-4 构建了一个显示渐变的网格，使用 `fx` 属性在中心点左右移动焦点，还可以使用 `cy` 属性上下移动渐变圆的中心点。组合结果如图 9-10 所示。

该例中设置 `spreadMethod` 的值为 `reflect` 来显示各个方向上重复距离是如何与焦点和圆的边之间的原距离相匹配的。这看起来真的很酷！

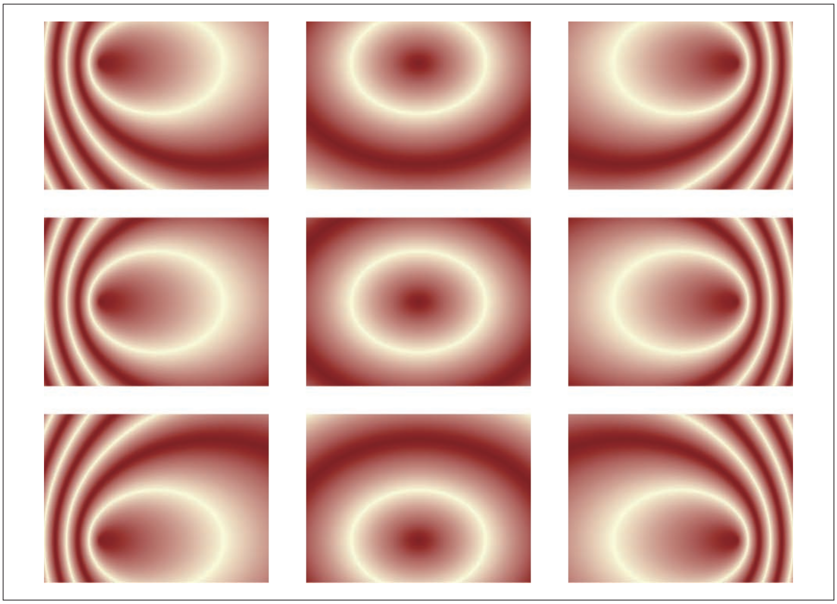


图 9-10：比较径向渐变中心点改变（行）和焦点位置改变（列）的效果

例 9-4　控制径向渐变的尺寸和位置

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="3in">
    <title xml:lang="en">Focus Point Versus Center Point</title>
    <defs>

        <radialGradient id="center" spreadMethod="reflect"
                        r="30%">                                  ❶
            <stop stop-color="darkRed" offset="0"/>
            <stop stop-color="lightYellow" offset="1"/>
        </radialGradient>
        <radialGradient id="left" xlink:href="#center"
                        fx="25%"/>                                ❷
        <radialGradient id="right" xlink:href="#center"
                        fx="75%"/>

        <radialGradient id="left-up" xlink:href="#left"
                        cy="25%"/>                                ❸
        <radialGradient id="left-down" xlink:href="#left"
                        cy="75%"/>
        <radialGradient id="center-up" xlink:href="#center"
                        cy="25%"/>
```

```
        <radialGradient id="center-down" xlink:href="#center"
                        cy="75%"/>
        <radialGradient id="right-up" xlink:href="#right"
                        cy="25%"/>
        <radialGradient id="right-down" xlink:href="#right"
                        cy="75%"/>

        <rect id="r" width="30%" height="30%" />          ❹
    </defs>
    <use xlink:href="#r" x="0%"  y="0%"  fill="url(#left-up)"/>
    <use xlink:href="#r" x="35%" y="0%"  fill="url(#center-up)"/>
    <use xlink:href="#r" x="70%" y="0%"  fill="url(#right-up)"/>
    <use xlink:href="#r" x="0%"  y="35%" fill="url(#left)"/>
    <use xlink:href="#r" x="35%" y="35%" fill="url(#center)"/>
    <use xlink:href="#r" x="70%" y="35%" fill="url(#right)"/>
    <use xlink:href="#r" x="0%"  y="70%" fill="url(#left-down)"/>
    <use xlink:href="#r" x="35%" y="70%" fill="url(#center-down)"/>
    <use xlink:href="#r" x="70%" y="70%" fill="url(#right-down)"/>
</svg>
```

❶ 将被用于中间部分的渐变使用默认居中的圆形和焦点。半径设置为对象边界盒的 30%，给重复效果留下空间。

❷ 中间一行的其他渐变通过调整 `fx` 在中心的左侧或右侧来创建，通过使用 `xlink:href` 属性来复制第一个渐变的颜色结点和半径。

❸ 剩下的渐变从前面的某个渐变中复制颜色结点、半径以及焦点，之后调整 `cy` 的值。

❹ 因为我们要有 9 个同样宽高的长方形，所以预先定义一个形状，然后使用 `<use>` 元素来复制。

如你所见，改变渐变圆形的中心点会变换整个渐变图案的位置。相比而言，改变焦点会明显改变渐变的整体图案，尤其是在重复渐变中。

> 如果你只改变中心点而没有改变焦点，当然会大大改变渐变的外观。例 9-4 中，没有设置渐变的垂直位置 `fy`，所以它可以自动调整来匹配 `cy` 的变化。

聚焦未来

从一个点扩展聚焦区域

SVG 2 中给径向渐变增加了一个新的 `fr` 属性。它将会围绕焦点定义一个圆，表示渐变 0% 偏移的位置。相似的参数已经可以在 HTML canvas 绘图

函数中使用。

当焦点在主要的圆的中心时，渐变的结果与第一个结点偏移不为零时一样，除非有额外的间隔没有包含在循环周期内。对于不对称渐变，它的几何形状将与任何现在可以通过 SVG 生成的形状都不同。

9.5 变换径向渐变

gradientTransform 属性也可以用在 <radialGradient> 上，且功能与用在线性渐变上完全相同，它会改变渐变底层坐标系，平移、旋转、缩放或倾斜不仅仅作用在中心点而且是作用在整个渐变上。

再次说明，你必须要注意拉伸对象边界盒坐标系的效果。无论如何旋转 objectBoundingBox 径向渐变，它总会被沿着形状的长轴拉伸。

但是你可以通过使用 skew 变换来创建一个沿不同轴拉伸的椭圆。例 9-5 在焦点不同的三个渐变上使用相同的 skew 变换，结果如图 9-11 所示。

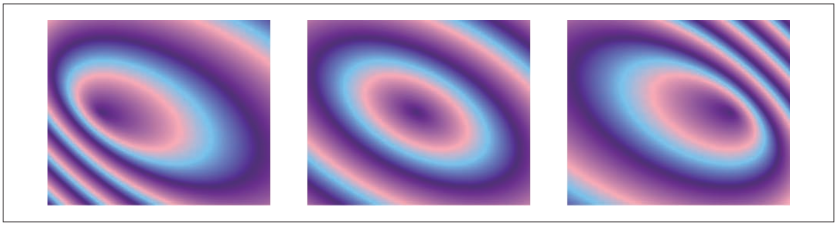


图 9-11：倾斜焦点不同的径向渐变

例 9-5 倾斜径向渐变

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="1in">
    <title xml:lang="en">Skewed Radial Gradients</title>
    <defs>

        <radialGradient id="center" spreadMethod="repeat" r="40%"
            gradientTransform="translate(-0.3,0) skewX(30)">    ❶
            <stop stop-color="indigo" offset="0"/>
            <stop stop-color="lightPink" offset="0.5"/>
            <stop stop-color="lightSkyBlue" offset="0.7"/>
            <stop stop-color="indigo" offset="1"/>              ❷
        </radialGradient>
```

```
        <radialGradient id="left" xlink:href="#center"
                        fx="25%"/>
        <radialGradient id="right" xlink:href="#center"
                        fx="75%"/>                           ❸

        <rect id="r" width="30%" height="100%" />
    </defs>

    <use xlink:href="#r" x="0%"  y="0%" fill="url(#left)"/>
    <use xlink:href="#r" x="35%" y="0%" fill="url(#center)"/>
    <use xlink:href="#r" x="70%" y="0%" fill="url(#right)"/>
</svg>
```

❶ `skewX(30)` 变换将会把渐变上的点向右移动 *y* 轴增加的距离。

❷ 渐变使用的是 `repeat` 扩展方式，所以结点的顺序将不会改变。第一个和最后一个结点颜色相同，这就避免了锐利的过渡。

❸ 与例 9-4 相同，其他的渐变是通过左右移动焦点创建的。

例 9-5 中使用的 `gradientTransform` 中还包括一个水平的变换来重新设定渐变的中心位置，它也会受到 `skew` 的影响。

`<radialGradient>` 上的变换是相对于坐标系的原点（左上角）而不是渐变的中心来计算的。

倾斜和偏离中心的焦点使得径向渐变不局限于完美的几何对称，还增加了三维的感觉。因此，这也是渐变可以用在写实绘制中的一种方式。

9.6 大型渐变

要想细致入微地表现真实对象，你需要考虑光线和阴影的颜色变化。SVG 包含了滤镜函数，用它来模拟光照效果，但是这会增加额外的处理时间。如果你可以使用渐变来达到同样的效果，往往会更加高效，尤其是在你之后想要给图片添加动画的时候。

图 9-12 比较了这两种效果。一个形状使用不对称的径向渐变来模拟黄色光照射在一个红色球上的效果，另一个使用光照滤镜来实现相似的效果。渐变不能创建像滤镜那样的三维曲线，但非常接近。完整的代码如例 9-6 所示。

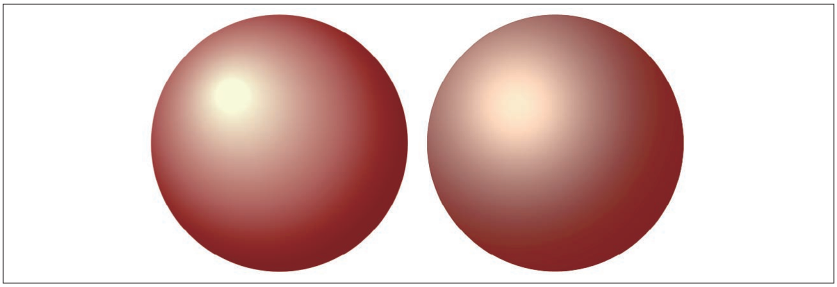


图 9-12：聚光灯照在球上，通过近似的渐变生成（左）或光照滤镜计算生成（右）

例 9-6　使用渐变或滤镜创建灯光效果

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4in" height="2in" >
    <title xml:lang="en">Simulated Lighting Versus
                        Lighting Filters</title>

    <radialGradient id="faux-lighting"
                    cx="45%" cy="45%" r="60%"
                    fx="30%" fy="30%" >                    ❶
        <stop offset="0.1" stop-color="lightYellow" />
        <stop offset="0.9" stop-color="darkRed" />         ❷
    </radialGradient>
    <circle cx="25%" cy="50%" r="0.9in"
            fill="url(#faux-lighting)"/>                   ❸
    <filter id="yellow-glow" primitiveUnits="objectBoundingBox">
        <feGaussianBlur in="SourceAlpha" stdDeviation="0.3" />
        <feComposite in2="SourceAlpha" operator="in" />    ❹
        <feSpecularLighting surfaceScale="0.1"
                     specularConstant="1"
                     specularExponent="20"
                     lighting-color="lightYellow"
                     result="highlight">
            <fePointLight x="0.35" y="0.35" z="0.7"/>      ❺
        </feSpecularLighting>
        <feComposite in="SourceGraphic" in2="highlight"
                     operator="arithmetic"
                     k1="0" k2="1" k3="0.9" k4="0" />
        <feComposite in2="SourceAlpha" operator="in" />    ❻
    </filter>
    <circle cx="75%" cy="50%" r="0.9in"
            fill="darkRed" filter="url(#yellow-glow)" />   ❼
</svg>
```

❶ <radialGradient> 定义了一个比边界盒稍大的圆，并向左上方偏离中心。焦点在左上方移动距离更多的地方。因此形状的这一边会受到初始颜色结点更大的影响，而右下方将被最后的颜色结点控制。

❷ 结点与渐变射线的起点和终点略有偏移，在焦点周围创建了一个纯黄光的小圆，并在渐变圆的边上露出最后的红色。

❸ 在第一个圆上简单地应用渐变填充的效果。

❹ 滤镜首先生成形状 alpha 通道的一个模糊版本，然后把它沿着原形状的边缘裁剪。这时就创建了一个距形状边缘越近，alpha 的值越小的层。

❺ 光照滤镜使用上一步的 alpha 通道作为隆起映射来定义对象的三维形状。然后计算添加到形状上的光的量，使用各种各样的属性来定义形状的反射效果，使用 `<fePointLight>` 元素来定义入射光的位置和图案。`lighting-color` 属性指定了浅黄色的光。

❻ 最后一步把光照效果和底层彩色图形结合在一起，并再次裁剪它，所以光只在形状最初不透明的部分可见。

❼ 第二个圆使用纯 `darkRed` 色填充，然后通过 `filter` 属性应用光照效果。

目前使用渐变代替光照滤镜最充分一个理由就是跨浏览器的一致性。即使所有浏览器都实现了光照滤镜，但是浏览器之间的实现并不一致。

> WebKit 目前没有实现光照效果的滤镜，该滤镜元素会返回一个透明层；结合例 9-6 中的例子来说，结果显示的是原始的平坦的红色圆圈；与其他滤镜结合，图形可能会完全消失。

甚至在支持滤镜光照效果的浏览器中，真切的执行效果也有很大的不同。图 9-12 中显示的是在 Firefox 中的效果；图 9-13 是相同的代码在 Chrome 中的渲染效果，光照更加强烈；IE 会生成介于两者之间的一个图片。

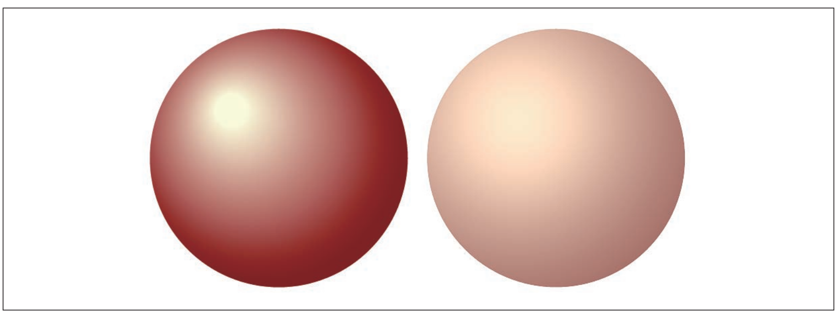


图 9-13：聚光灯照在球上在另一种浏览器中的效果，左侧是通过渐变生成，右侧是通过滤镜生成

选择渐变而非滤镜的另一个原因是性能问题，有两方面的原因。第一，光照滤镜需要通过多次计算而渐变通过单个函数来生成。第二，滤镜在渲染后、光栅化后的图片上进行计算，所以每次底层形状改变或移动时，它都需重新计算。

CSS 与 SVG
使用 CSS 径向渐变模仿焦点

目前在 CSS 中还没有办法调整渐变焦点。某些情况下，对于非重复渐变，你可以使用分层的背景图片来模拟焦点不对称的效果。焦点颜色使用从有色到透明的一个或多个渐变来绘制，然后把它放在代表最终形状的渐变之上。

基于这种方法，例 9-7 使用 CSS 渐变重新创建了聚光灯照在红色球上的效果。图 9-14 展示了与原 SVG 渐变版本对比的结果。

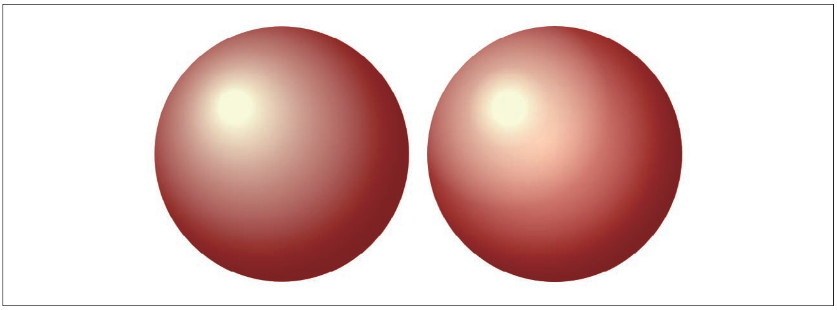


图 9-14：使用一个单独的 SVG 渐变（左）和分层的 CSS 渐变（右）来生成焦点渐变效果

例 9-7　使用分层的 CSS 渐变模拟偏离中心的放射光

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>CSS and SVG Simulated Lighting Gradients</title>
    <style type='text/css'>
    html, body {
        height: 2in;
        width: 4in;
        margin: 0; padding: 0;
        text-align: center;                                    ❶
    }
    svg, div.sphere {
```

```
            height: 1.8in;
            width: 1.8in;
            display: block;
            float: left;
            margin: 0.1in;
        }
        div.sphere {
            border-radius: 50%;
            background:
                radial-gradient(ellipse 40% 40% at 32% 32%,
                    lightYellow 15%,
                    rgba(100%, 100%, 88%, 0.5) 40%,
                    transparent),
                radial-gradient(ellipse 55% 55% at 40% 40%,
                    rgba(100%, 100%, 88%, 0.4) 20%,
                    transparent),
                radial-gradient(ellipse 60% 60% at 45% 45%,
                    rgb(100%,55%,45%), darkRed 90%);            ❷
         }
        </style>
    </head>
    <body>
        <svg>
            <radialGradient id="faux-lighting"
                            cx="45%" cy="45%" r="60%"
                            fx="30%" fy="30%" >
                <stop offset="0.1" stop-color="lightYellow" />
                <stop offset="0.9" stop-color="darkRed" />
            </radialGradient>
            <circle cx="50%" cy="50%" r="50%"
                    fill="url(#faux-lighting)"/>                ❸

        </svg>
        <div class="sphere"></div>                              ❹
    </body>
    </html>
```

❶ 前两个 CSS 规则对网页整体进行布局，使它与例 9-6 中的 SVG 布局相匹配，包括（通常不推荐）整个网页的尺寸。

❷ 渐变效果作为分层的背景图像来使用，结合三个单独的 radial-gradient 函数来创建把焦点移动到圆边缘的效果。中间的渐变的结点使用 rgba 函数创建了部分透明的 lightYellow 色。

❸ 作为对比的 SVG 渐变使用内联的 SVG 代码创建。

❹ 单独使用一个 <div> 标签来显示 CSS 渐变生成的背景内容。

为了在所有浏览器中都有最佳效果，例 9-7 中使用的 `transparent` 关键词应该通过表示亮一些的浅黄色的 `rgba` 函数来代替。一些旧版本的 Firefox 在向 `transparent` 关键词过渡时，会将颜色转向黑色（官方的透明黑色）。该规范现在提供了明确的引导方案来避免这种问题。

目前已经提议给 CSS 渐变添加焦点参数（包括焦点半径）来更简单地创建这种效果，但具体语法还没有最终确定。

SVG 渐变是一个看似简单的话题。它只有两种类型的渐变，线性和径向，但是它有无数可能的变化来创建不同的效果。通过这些变化，渐变可以给简单的矢量形状添加细微差别和小细节。

渐变是 SVG 开发者工具包的一个主要部分，并且可被用于创建非常复杂和精密的图像。在大多数情况下，这些图片不是通过单个精确的几何渐变组成，而是由多个部分透明渐变的元素重叠而成。

我们可以想象一个戏剧舞台之类的东西来做个练习，完整的场景包括侧面红色的天鹅绒、顶部的幕布以及配有聚光灯的舞台。作为图片，这可以方便地突出给定的对象（图像、幻灯片、视频以及文本块），并且由于它是 SVG，我们可以直接对幕布进行打开关闭操作。

当幕布打开或收起时，它会操作可见光来显示明亮和阴影区域，这可以通过一个有阴影的红色渐变来很好地实现。我们中的 Amelia 之前使用简单重复渐变填充的长方形创建了一个类似的场景。图 9-15 显示的场景是由 Kurt 创建的，他通过谨慎地控制渐变结点，使它的位置匹配路径曲线来遮挡幕布。

该图片由三个不同的层构成。

后面覆盖整个宽度的是舞台，它是一个简单的长方形，且渐变从距离顶部 350 个单位的位置开始，它上面有多个渐变叠加来创建聚光灯照射在木地板上的效果。舞台的边其实是舞台中一个渐变的一部分，它还告诉你如何使用这种渐变来“画”难一点儿的边界。反过来说，光照大部分都是透明的，距离中心越近，不透明度越高，焦点偏移是为了创建聚光灯从一个角度照射在舞台上的效果。这是一个柔和的聚光灯，我们没有明显地把它的边刻画出来，而是隐藏在周围的黑暗中。

下一层由左侧和右侧的幕布组成，它们使用了相同核心路径创建的形状和一个非常复杂的使用褶皱来表示光照和阴影的红色 / 红褐色图案。右侧的幕

布与左侧的完全一样，只是进行了翻转并转换到与屏幕的右侧对齐。每一侧的幕布后面都有一个幕布阴影，它们是由相同的路径生成但略微有一些偏移，并且都有 50% 的不透明度。

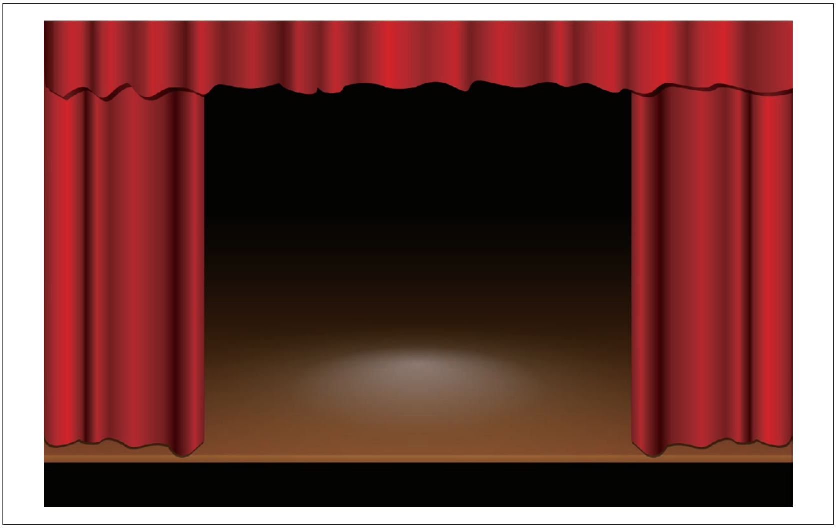


图 9-15：使用 SVG 创建悬挂幕布的舞台

最后，顶部的幕布覆盖在其他幕布之上，且就像侧面幕布那样使用非常复杂的有 40 个结点的线性渐变填充，这是通过试验和排错确定的。它同样有一个偏移的阴影。

该图片完整的代码如例 9-8 所示。

例 9-8　使用复杂且分层的渐变创建一个舞台

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%" viewBox="0 0 1000 650"
     id="curtains" >                                   ❶
    <title xml:lang="en">Stage Curtains</title>
    <style type="text/css">
        .curtain-shadow {
            fill: black;
            fill-opacity: 0.5;                         ❷
        }
        .side-curtain {
            fill: url(#side-curtain-gradient);
        }
```

```
        .ceiling-curtain {
            fill: url(#ceiling-curtain-gradient);
        }
    </style>
    <defs>                                                    ❸
        <path id="curtain" transform="scale(0.55)"
              d="m 0.319,0 0.252,1000 c 0,0 -2.761,40 44.129,30
46.9,-10 38.6,-20 46.9,-10 8.3,0 27.4,10 47.4,0 19,-21 63,10 63,10
0,0 11,10 44,0 33,-10 58,0 58,0 0,0 19,40 52,20 33,-30 34,-30
34,-30 L 390,-0.53 z"/>
        <path id="ceiling-curtain" transform="scale(1,0.5)"
              d="m 0,0 1000,0 0,180 c 0,0 -26,17 -49,9 -22,-9
-48,-32 -57,-18 -8,15 -31,3 -54,-5 -23,-9 -20,45 -49,23 -28,-23
-11,11 -34,0 -23,-12 -23,-32 -46,-18 -22,15 -17,-8 -37,-8 -20,0
-25,23 -54,11 -29,-11 -43,-25 -49,-3 -5,23 -5,23 -28,0 -23,-22
-46,6 -46,6 0,0 -28,12 -43,-3 -14,-14 -54,-8 -54,6 0,14 -23,20
-31,3 -9,-17 8,26 -29,8 -37,-17 -11,-34 -40,-17 -29,17 -49,-3
-63,-5 -14,-3 -14,37 -28,25 -15,-11 -9,-14 -32,-17 -23,-3 -28,37
-46,23 C 114,186 120,157 106,169 91.4,180 85.7,214 74.3,203
62.9,191 60,171 48.6,177 37.1,183 25.7,203 25.7,203 L 2.86,177 C
0,171 0,0 0,0 z"/>
        <linearGradient id="side-curtain-gradient">           ❹
            <stop stop-color="#800000" offset="0"/>
            <stop stop-color="#c00000" offset="0.08"/>
            <stop stop-color="#e00000" offset=".15"/>
            <stop stop-color="#800000" offset="0.24"/>
            <stop stop-color="#400000" offset="0.26"/>
            <stop stop-color="#600000" offset="0.28"/>
            <stop stop-color="#800000" offset="0.30"/>
            <stop stop-color="#c00000" offset="0.33"/>
            <stop stop-color="#800000" offset="0.41"/>
            <stop stop-color="#c00000" offset="0.56"/>
            <stop stop-color="#800000" offset="0.75"/>
            <stop stop-color="#400000" offset="0.82"/>
            <stop stop-color="#800000" offset="0.89"/>
            <stop stop-color="#C00000" offset="0.94"/>
            <stop stop-color="#800000" offset="1"/>
        </linearGradient>
        <linearGradient id="ceiling-curtain-gradient">
            <stop stop-color="#400000" offset="0"/>
            <stop stop-color="#700000" offset="0.013"/>
            <stop stop-color="#800000" offset="0.013"/>
            <stop stop-color="#C00000" offset="0.03"/>
            <stop stop-color="#D00000" offset="0.04"/>
            <stop stop-color="#C00000" offset="0.05"/>
            <stop stop-color="#600000" offset="0.065"/>
            <stop stop-color="#B00000" offset="0.078"/>
            <stop stop-color="#D00000" offset="0.09"/>
```

```
        <stop stop-color="#800000" offset="0.12"/>
        <stop stop-color="#A00000" offset="0.135"/>
        <stop stop-color="#D00000" offset="0.145"/>
        <stop stop-color="#C00000" offset="0.16"/>
        <stop stop-color="#600000" offset="0.19"/>
        <stop stop-color="#D00000" offset="0.21"/>
        <stop stop-color="#800000" offset="0.24"/>
        <stop stop-color="#C00000" offset="0.28"/>
        <stop stop-color="#600000" offset="0.32"/>
        <stop stop-color="#800000" offset="0.33"/>
        <stop stop-color="#C00000" offset="0.34"/>
        <stop stop-color="#800000" offset="0.37"/>
        <stop stop-color="#D00000" offset="0.39"/>
        <stop stop-color="#800000" offset="0.42"/>
        <stop stop-color="#E00000" offset="0.46"/>
        <stop stop-color="#A00000" offset="0.51"/>
        <stop stop-color="#D00000" offset="0.55"/>
        <stop stop-color="#800000" offset="0.57"/>
        <stop stop-color="#D00000" offset="0.61"/>
        <stop stop-color="#800000" offset="0.67"/>
        <stop stop-color="#D00000" offset="0.69"/>
        <stop stop-color="#800000" offset="0.71"/>
        <stop stop-color="#C00000" offset="0.74"/>
        <stop stop-color="#D00000" offset="0.76"/>
        <stop stop-color="#800000" offset="0.78"/>
        <stop stop-color="#C00000" offset="0.80"/>
        <stop stop-color="#E00000" offset="0.83"/>
        <stop stop-color="#800000" offset="0.86"/>
        <stop stop-color="#C00000" offset="0.88"/>
        <stop stop-color="#E00000" offset="0.91"/>
        <stop stop-color="#800000" offset="0.94"/>
        <stop stop-color="#C00000" offset="1"/>
    </linearGradient>
    <linearGradient id="stageGradient" x2="0" y2="1">  ❺
        <stop stop-color="#100800" offset="0"/>
        <stop stop-color="saddleBrown" offset="0.97"/>
        <stop stop-color="#A06020" offset="0.97"/>
        <stop stop-color="saddleBrown" offset="1"/>
    </linearGradient>
    <linearGradient id="shadowGradient" x2="0" y2="1"> ❻
        <stop stop-opacity="0.6" offset="0" />
        <stop stop-opacity="0" offset="0.97"/>
    </linearGradient>
    <radialGradient id="spotlightGradient" cy="0.9" fy="0.6">
        <stop stop-color="#FFFFFF" offset="0"
              stop-opacity="0.4"/>
        <stop stop-color="#FFFFFF" offset="0.35"
              stop-opacity="0.1"/>
```

```
            <stop stop-color="#000000" offset="1"
                  stop-opacity="0"/>                    ❼
        </radialGradient>

        <rect id="stage" width="1000" height="330" y="260" />
    </defs>

    <rect id="background" width="100%" height="100%"
          fill="black"/>

    <g id="stage-illuminated">                          ❽
        <use xlink:href="#stage" fill="url(#stageGradient)"/>
        <use xlink:href="#stage" fill="url(#shadowGradient)"/>
        <use xlink:href="#stage" fill="url(#spotlightGradient)"/>
    </g>

    <g id="side-curtain-left" >                         ❾
        <use xlink:href="#curtain" class="curtain-shadow"
             x="-1" y="3"/>
        <use xlink:href="#curtain" class="side-curtain"/>
    </g>
    <g id="side-curtain-right"
       transform="translate(1000,0) scale(-1,1)">       ❿
        <use xlink:href="#side-curtain-left"/>
    </g>
    <g id="ceiling-curtain">                            ⓫
        <use xlink:href="#ceiling-curtain" class="curtain-shadow"
             x="5" y="5" />
        <use xlink:href="#ceiling-curtain"
             class="ceiling-curtain"/>
    </g>
</svg>
```

❶ <svg> 元素上的 id 使整个场景可以在另一个图片（虽然你必须把标记导入到其他文件中使所有样式能够在 Web 浏览器中正常工作）中重复使用。

❷ 幕布的 fill 值使用 CSS 规则来设置。特别是幕布的阴影，它是简单地通过使用半透明的黑色渐变进行填充来代替的。

❸ 两个复杂的 path 元素，一个是侧边的幕布，另一个是天花板下幕布的边缘。它们都没有直接设置样式属性，而是在使用 <use> 元素复制时添加样式。

❹ 表示幕布的前两个 <linearGradient> 对象设计过程中尽量与 <path> 元素匹配。许多 SVG 图片编辑器都有可视化工具来使这项工作更加容易，当然有一些不足之处也是不可避免的。

❺ 剩下的渐变用于构建舞台；第一个是从上到下填充木制的颜色，在底部绘制舞台边缘的地方有一个锐利的过渡。

❻ shadowGradient 使用默认的黑色 stop-color，然后调整 stop-opacity 而不是创建额外的层作为舞台从后到前（长方形从上到下）的阴影。把它应用为单独的一层是为了方便之后通过色相来调整亮度。

❼ 创建聚光灯的 <radialGradient> 同样布局在舞台之上，所以它同样使用了 stop-opacity 属性来允许下面的层露出来。

❽ 在绘制一个纯黑色背景之后，舞台还使用相同的长方形绘制三次，来布局三个不同的渐变。这个完美的使用分层填充的例子表达了对 SVG 2 的功能的迫切期待——它将允许舞台绘制在单独的形状上，然后三个渐变一起填充。

❾ 左侧的幕布是通过一起布局两个版本的幕布形状进行绘制，一个添加了阴影样式，另一个是幕布的主要样式。

❿ 右侧的幕布通过复制和转换左侧的幕布来绘制。

⓫ 最后，天花板上的幕布使用同样的布局方法绘制。

虽然例 9-8 中包含许多代码，但相较于本书中的大多数例子，它作为图片文件依然是相对较小的：可编辑的文本有 7KB，通过 gzip 压缩之后只剩下 1.4KB。相比而言，相同图片的 PNG 版本，要想填满桌面显示的大部分区域，要超过 80KB 的大小。

聚焦未来

颜色弹性过渡的网格渐变

舞台与幕布的例子既优雅，也非常便于使用 SVG 渐变。悬挂窗帘的褶子正好是平行的，所以正好使用线性渐变。但在普遍情况下，这样的描述往往并不准确，对象的颜色和阴影会在各个方向上进行过渡，而不只是平行线或椭圆。目前，不管重建何种类别的复杂阴影图案，你只能同时布局多种不同的半透明渐变。

SVG 2 中引入了一个描述复杂二维颜色过渡的方法，我们称之为网格渐变。网格是直线或曲线交叉形成的格子。渐变颜色的结点被分配到这些路径的交点上，渐变的每一片（四条曲线围起来的区域）的颜色也是基于它们的值来插入。你可能不会通过手动编码来生成网格渐变，Adobe Illustrator 和 Inkscape 已经支持渐变网格的创建，并且 SVG 提案的设计也是兼容它们的。

网格渐变太复杂了，CSS 还没有与之对应的候选渐变函数。但是，CSS Image Values and Replaced Content Module Level 4 中引入了一个更加易于管理的新的渐变函数 conic-gradient()，以及它的变形 repeating-conic-gradient()。

锥形渐变是一种颜色会随着你围绕中心点做圆周运动而改变的渐变，每一种颜色都沿着从中心点发散出来的射线尽可能地扩展 。不同于径向渐变，径向渐变的颜色是沿着焦点发散的射线而改变，而在同心圆上是一样的。锥形渐变的结点不是通过固定的距离而定义，而是通过固定的角度来定义。

提议的 CSS 语法将使用 at 关键词（可选）来描述中心点的位置（默认情况下，是图片的中心），这与径向渐变的语法相同。接着是结点的列表，偏移使用角度单位或整个圆的百分比来描述。零度表示的是从中心点朝上的垂直线。0° ～ 360° 之外的角度将会被裁剪。与其他 CSS 渐变一样，没有定义偏移值的颜色结点将均匀分配。

下面列出了一些合法的锥形渐变语法：

```
/* a hue color wheel */
conic-gradient(red,yellow,lime,cyan,blue,magenta,red);

/* oscillating orange rays */
repeating-conic-gradient(at top left,
               tomato, gold 10deg);

/* checkerboard */
conic-gradient(black 0 25%, white 25% 50%,
               black 50% 75%, white 75% 100%);
```

最后一个例子使用另一种在 4 级规范中介绍的简写方式，它将适用于所有的 CSS 渐变函数。当一个渐变包括一个纯色区域时，你可以列出单独颜色值的两个偏移，而不一定非要定义连续两个颜色相同的结点。

编写本书时，锥形渐变在任何浏览器中都没有直接支持。Lea Verou，CSS 工作组的一员，创建了一个 JavaScript polyfill，使用 HTML canvas 把锥形渐变的声明转变为静态图片。

因为 CSS 渐变函数将可以直接在 SVG 2 中使用，所以目前没有创建专门的 SVG 锥形渐变元素的计划。但是相同的（或者非常接近的）效果可以使用网格渐变来创建。

第10章

磁贴与纹理

在第 5 章介绍渲染服务时，我们提到你可能希望使用多种不同的资源来绘制一个形状：单一的颜色、一个或多个渐变、重复图案、位图、文本，甚至是其他 SVG 文件。到目前为止，我们已经介绍了如何使用纯色或单个渐变来进行绘制。对于所有其他可能的选项，我们都将使用一个单独的渲染服务元素：`<pattern>`。

一个 SVG 图案定义了一个可作为其他形状的渲染服务的 SVG 图形块。我们可以使用任何 SVG 内容，包括图片、文本以及使用渐变填充的形状。图案是一种重复矩形（或转换后的矩形）磁贴（tile）的布局。你也可以制作一个形状相同大小的单个磁贴来创建一个没有重复的填充，这为我们提供了更多的可能，在第 11 章中将分别进行讨论。

有多种可选方案来设置磁贴和图案内容的大小。这使得许多不同的设计成为可能，但也有些依然很难实现。本章会试图讲解什么可以实现而什么无法实现，并且给一些不尽如人意的情况提供了变通方案。我们也比较了 SVG 图案和重复 CSS 背景图片，后者是许多 Web 设计师都熟悉的一种重复图案。

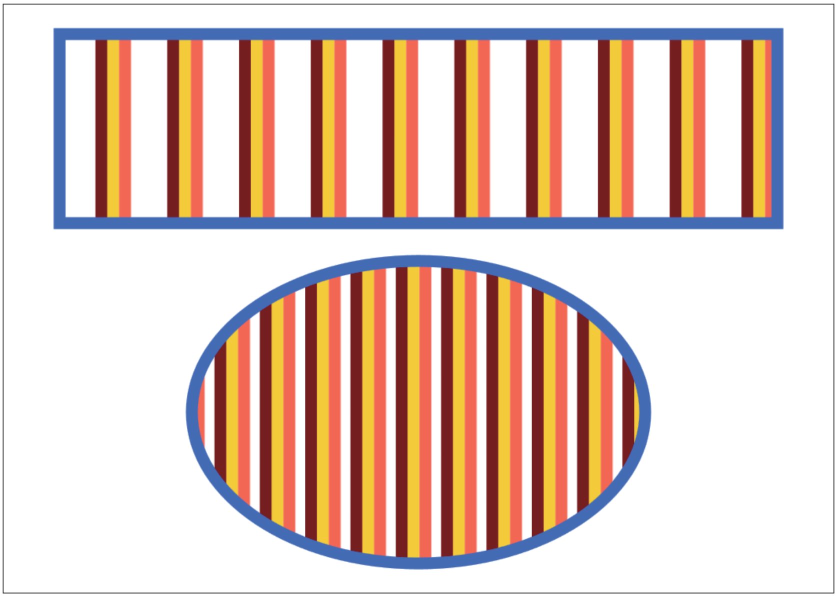


图 10-1：使用简单的重复条纹图案填充的形状

10.1 搭积木

许多时候，<pattern> 元素与前几章介绍的渐变很相似。元素本身的属性定义了形状以及图案磁贴（重复单元）的尺寸。<pattern> 的子内容组成了绘制到屏幕上的图形。

<linearGradient> 和 <radialGradient> 上的属性分别与 <line> 和 <circle> 匹配，而 <pattern> 上的几何属性与 <rect> 或 <image>：x，y，width 和 height 类似。它们的默认值都是 0，通常情况下，width 或 height 为 0 时内容不会被绘制。

为了理解图案是如何工作的，最好的做法是先运行一个示例。例 10-1 给出了一个简单的条纹图案的代码，并用它填充了两个形状。图 10-1 显示了运行结果。

例 10-1　使用简单的重复条纹图案填充形状

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="3in" >
```

```
        <title xml:lang="en">Striped Pattern</title>
        <style type="text/css">
            svg {
                stroke-width: 6px;
            }
        </style>
        <pattern id="stripes" x="5%" width="10%" height="100%">
            <line x1="3" x2="3" y2="100%" stroke="maroon" />
            <line x1="9" x2="9" y2="100%" stroke="gold" />
            <line x1="15" x2="15" y2="100%" stroke="tomato" />
        </pattern>

        <g stroke="royalBlue" fill="url(#stripes)">
            <rect x="0.1in" y="0.1in" width="3.8in" height="1in" />
            <ellipse cx="50%" cy="2.1in" rx="1.2in" ry="0.8in" />
        </g>
    </svg>
```

图案本身通过如下代码声明：

```
<pattern id="stripes" x="5%" width="10%" height="100%">
```

id 属性，和其他渲染服务一样是必需的，这样图案才能使用。width 和 height 属性定义了每个重复磁贴的尺寸。磁贴从水平偏移 x 和垂直偏移 y（这里使用默认的 0）的点开始排列。(x,y) 点定义了被引用的磁贴左上角的位置，剩下的磁贴之后再上下左右重复多次来填充形状。

那么 width、height、x、y 是如何计量的呢？再看图 10-1，注意两个形状的不同之处。两个形状条纹的宽度是一样的，但是椭圆中条纹之间距离更近。很明显宽度不是一个绝对值。

默认情况下，控制图案磁贴尺寸的属性是基于 objectBoundingBox 单位来计算的。每块磁贴占满边界盒的高度（100%），但宽度只占 10%。无论形状有多宽，它都会有 10 条间隔开的条纹。

根据渐变中的经验，你可能已经猜到磁贴默认的对象边界盒尺寸可以修改为 userSpaceOnUse。

相关的属性是 patternUnits，10.2 节将会讲解改变它后所产生的效果。

x 和 y 偏移也是基于边界盒来测量的。在例 10-1 中，5% 的水平偏移是 10% 的磁贴宽度的一半。这样第一个完整的条纹会从边界盒的左上角移动 5% 的

距离。但图案仍然会在所有的方向上进行重复：在椭圆中，你在左侧能看到另一个条纹的边缘，这是因为磁贴内条纹的宽度大于 5% 的偏移。

因此，条纹本身并没有基于对象边界盒按比例缩放。默认情况下，图案的内容是基于 `userSpaceOnUse` 坐标系（即实际上是被填充形状的坐标系）计量的。该坐标系的原点被变换到每个磁贴的左上角。

> 图案图形的缩放是受到 `<pattern>` 元素上的 `patternContentUnits` 属性控制的。它的默认值是 `userSpaceOnUse`，另一个可选值是 `objectBoundingBox`。

条纹（三个 `<line>` 元素）被扩展到 SVG 整个高度的 100%，比实际所需还要长得多。根据 SVG 1.1 的规范，为了有效地填充形状，图案应该使用 100% 的值。然而，正如第 7 章中提到的那样，当形状嵌套在使用不同渲染服务的 SVG 元素中时，用户空间单位在不同浏览器之间的实现是不一致的。

对于许多图案的设计，有必要使它超出图形的大小来确保整个磁贴都被填充，而不必关心要填充的形状的尺寸是多少。图案元素默认会设置 `overflow: hidden`，使图形被裁剪到磁贴的大小。

> SVG 1.1 和 2 中对于图案有意不定义使其可见的 `overflow` 值，以此来应对 SVG 1 中实现的不一致性。这意味着每个浏览器都可以用它们选择的方式去处理。因此建议一般不要去修改它的值。

例 10-2 给出了一个更加实际的例子，使用的图案会基于边界盒缩放，但条纹不会。它使用了图案把一个长方形分割为固定数量的相同块，每一个都有一个很细的网格线作为轮廓。这种覆盖的网格是图像编辑软件中的一个常用工具，这里它显示在一个非常简化的 Web 应用中，且覆盖在一个图片上面。

为了证实图案磁贴的缩放效果，照片的缩略图同样包含在它自己的网格中。缩略图上的网格更小一些，但是网格线的宽度相同。在这两个例子中，实际绘制的线比我们需要的宽得多，通过隐藏溢出部分来把它调整到适合的宽度。完整的效果如图 10-2 所示。

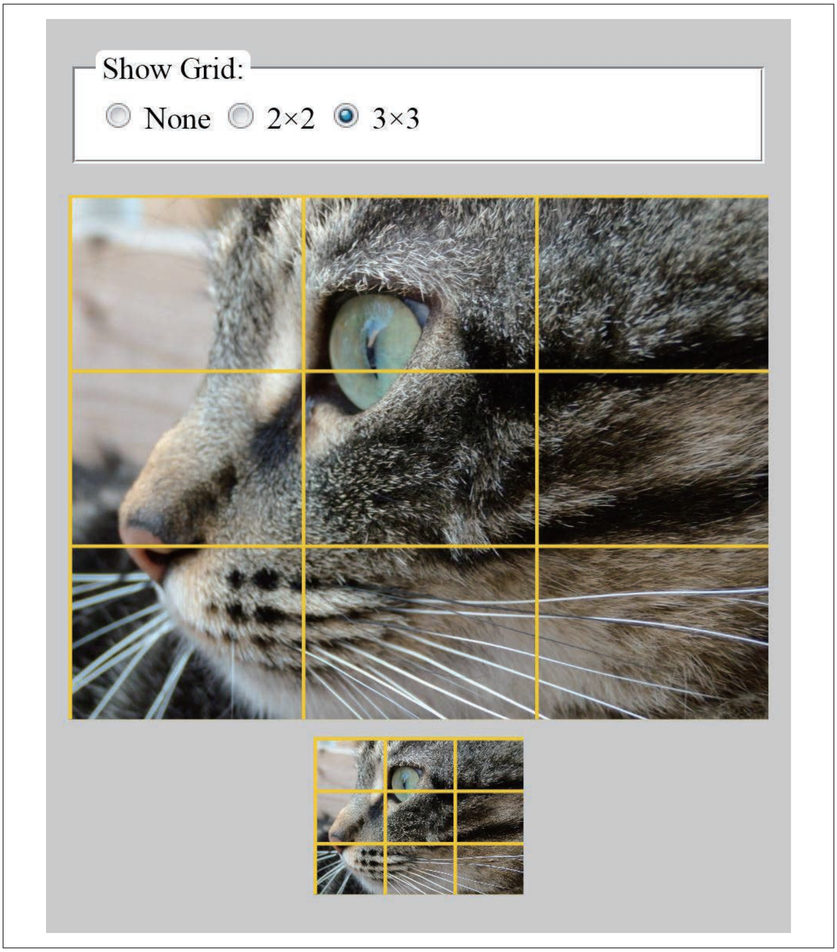


图 10-2：以相同的 3×3 的图案网格填充不同尺寸的长方形

例 10-2　图像编辑应用内的一个大小可变的网格图案

HTML 标记和 JavaScript：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Photo Grid Overlay</title>
    <style type='text/css'>
```

```
        /* styles must be in the same document */
    </style>
</head>
<body>
    <fieldset id="grid-options"
              role="radiogroup" aria-controls="graphics">
        <legend>Show Grid:</legend>                          ❶
        <label>
            <input type="radio" name="grid" value="none" checked>
            None</label>
        <label><input type="radio" name="grid" value="grid4">
            2&times;2</label>
        <label><input type="radio" name="grid" value="grid9">
            3&times;3</label>
    </fieldset>

    <svg id="graphics" viewBox="0 0 400 400">
        <title>Photo View</title>
        <pattern id="grid4" width="50%" height="50%" >      ❷
            <g stroke="gold" stroke-width="4px">
                <line x2="100%" />                           ❸
                <line y2="100%" />
            </g>
        </pattern>
        <pattern id="grid9" xlink:href="#grid4"
                 width="33.33333%" height="33.33333%" />    ❹

        <g>
            <title>Full image</title>
            <image width="400" height="300"
                   xlink:href="cat.jpeg" />
            <rect width="400" height="300" class="grid" /> ❺
        </g>
        <g transform="translate(140,310)">
            <title>Thumbnail</title>
            <image width="120" height="90"
                 xlink:href="cat.jpeg" />
            <rect width="120" height="90" class="grid" />
        </g>
    </svg>
    <script>
        document.getElementById("grid-options")
           .addEventListener("change", updateGrid);         ❻

        function updateGrid(e) {
            var svg = document.getElementById("graphics"),
                radio = e.target;
            if (radio.checked) {
```

```
                svg.setAttribute("class", radio.value);    ❼
            }
        }
    </script>
</body>
</html>
```

❶ 初始的 HTML 标记设置了一个小的表单 `field` 来用于选择网格。

❷ 内联 SVG 代码定义了 `<pattern>` 元素。图案使用默认的 `patternUnits` 设置：图案磁贴的宽和高是相对于边界盒的百分比。

❸ 然而它的内容是绘制在用户空间坐标系内的。它由两条线（组合在一起来应用通用的样式）组成，每条线都从原点开始，然后沿着水平或垂直方向延伸 SVG 宽或高的长度，通常这会超出图案磁贴的边缘。

❹ 第二个图案通过简写的方式定义，使用 `xlink:href` 引用了第一个图案，然后覆盖了 `width` 和 `height` 属性的值。和渐变一样，`xlink:href` 属性允许一个 `<pattern>` 元素作为其他元素的模板。

❺ 剩下的 SVG 代码描述的是可见内容：元素以及一个与之对应的作为网格容器的元素。

❻ 简短的 JavaScript 代码段用于监听单选按钮的改变。

❼ 一个简单的函数用于改变 `<svg>` 元素上的 `class`，来使用合适的 CSS 样式。在这个简单的例子中，`setAttribute` 方法用于取消之前设置的类而给它应用一个新的类。对于更加复杂的应用程序，在操作时你可能需要更加小心以不影响其他的类。

CSS 样式：

```
body {
    padding: 0.5em 0.25em;
    background-color: lightGray;
}
fieldset {
    display: block;
    background-color: white;
    margin-bottom: 1em;             ❶
}
legend {
    background-color: inherit;
    border-radius: 0.25em;
    padding: 0 0.2em;
}
```

```
svg {
    width: 100%;
    min-height: 300px;
    max-height: 100vh;                        ❷
    shape-rendering: crispEdges;
}
.grid {
    fill: none;
    pointer-events: none;                     ❸
}
.grid4 .grid {
    fill: url(#grid4);                        ❹
}
.grid9 .grid {
    fill: url(#grid9);
}
```

❶ 最初的几个 CSS 规则控制网页和表单元素的整体布局。

❷ `<svg>` 元素设置了最小高度和最大高度，但是在最新的浏览器中它会根据 `viewBox` 属性设置的宽高比来自动调整大小。`shape-rendering: crispEdges` 设置关闭矢量图形的抗锯齿效果，否则可能会使网格线看起来比较模糊。

❸ 添加 `grid` 类的长方形将默认不进行填充。`pointer-events` 使它无论有没有填充都没有交互效果。

❹ 当 `grid4` 或 `grid9` 类添加到父元素上时，网格矩形将使用合适的图案来填充，通过一个匹配的 `id` 来区分。

最终的效果满足了基本功能，但是并不理想。网格线使用 4px 的描边宽度，但是一半宽度将被裁剪掉，因为描边的中心在网格磁贴的边缘线上。这就意味着我们只能看到矩形上边和左边的网格线，而看不到右边和下边的网格线。因此交点没有完全匹配到照片宽高在几何上的平分点或三等分点。

SVG 图案中没有可以固定它的简单方式。当我们给磁贴尺寸使用对象边界盒单位时，你无法通过给磁贴设置固定像素的偏移来弥补这种不平衡，因为 `x` 和 `y` 也是基于边界盒单位解析的。并且由于网格线是基于用户空间单位绘制的，所以没有办法在每个磁贴相对的边上设置额外的线。

10.2　适当拉伸

基于默认的 `objectBoundingBox` 单位的图案磁贴和 `userSpaceOnUse` 单位的图

案内容设计的能够工作良好的图案只占一小部分。对于除了细条纹和网格的图案，你通常希望图形缩放到与图案磁贴相匹配的大小。换句话说，你希望的是 patternUnits 和 patternContentUnits 这两个属性有相同的值，objectBoundingBox 或 userSpaceOnUse 都可以。

> 由于默认值不匹配，你通常只需要声明其中一个。设置 patternUnits = "userSpaceOnUse" 或 patternContentUnits = "objectBoundingBox"。

当图案磁贴和它的内容都是基于对象边界盒缩放时，你会创建一个能适应被填充形状的图案。每个盒子中的磁贴的数量通常都是相同的，整个图案会一起放大或缩小。

当你使用对象边界盒单位时，要记住第 7 章中坐标系被扭曲的教训。如果盒子不是正方形，坐标系将变得不均匀，水平单位和垂直单位会不相等。圆、文本、图片都会被拉伸，旋转的角度也不均匀。还要注意的是所有的一切都会根据新的单位来缩放，包括描边宽度和字体大小。

此外，对于 patternContentUnits，不同于图案磁贴的尺寸，它的百分比值不可以与小数互换。基于用户空间坐标系定义的 100% 会按比例放大到其他单位的缩放效果。

> 无论使用什么缩放方式，图案内容的百分比值都不会参考图案磁贴。

如果主视窗宽 200 个单位（1% 是 2 个水平单位）高 150 个单位（1% 是 1.5 个垂直单位），在缩放的坐标系中 1% 不是盒宽的 2 倍、盒高的 1.5 倍。在不同的 SVG 中，比例是不相同的。

它通常没有什么用处，所以当图案内容使用对象边界盒单位时，要尽量避免使用百分比。

当然，你可以使用小数，此时它是相对于边界盒宽或高的比例。

例 10-3 使用这种方法来调整例 10-1。现在彩色条纹的宽度和间距都会被缩放来适应不同的形状，如图 10-3 所示。

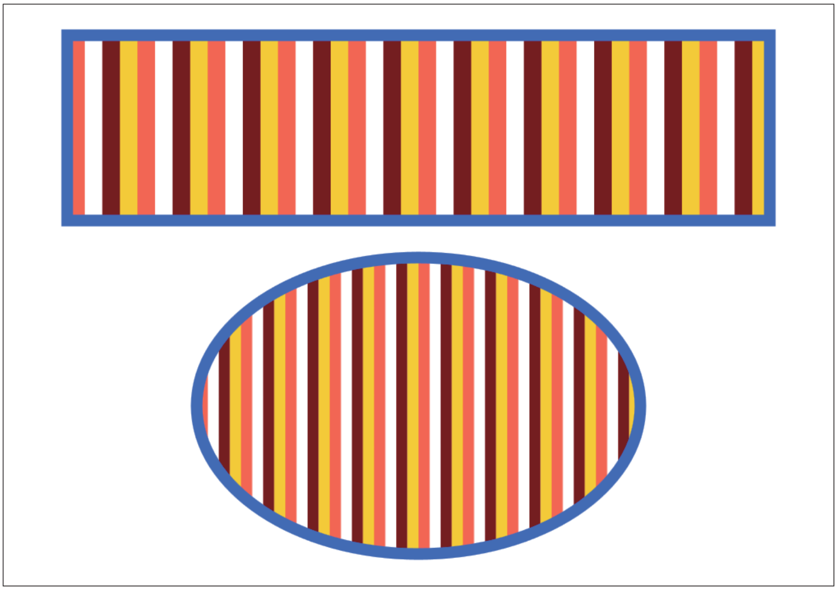


图 10-3：基于边界盒缩放的重复条纹图案填充的形状

例 10-3　使用缩放的重复条纹图案填充形状

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="3in" >
    <title xml:lang="en">Bounding Box Striped Pattern</title>
    <style type="text/css">
        svg {
            stroke-width: 6px;
        }
    </style>
    <pattern id="stripes2" x="5%" width="10%" height="100%"
             patternContentUnits="objectBoundingBox"
             stroke-width="0.025px" >                               ❶
        <line x1="0.0125" x2="0.0125" y2="1" stroke="maroon" />
        <line x1="0.0375" x2="0.0375" y2="1" stroke="gold" />       ❷
        <line x1="0.0625" x2="0.0625" y2="1" stroke="tomato" />
    </pattern>
    <g stroke="royalBlue" fill="url(#stripes2)">
        <rect x="0.1in" y="0.1in" width="3.8in" height="1in" />
        <ellipse cx="50%" cy="2.1in" rx="1.2in" ry="0.8in" />
    </g>
</svg>
```

❶ 在图案上明确设置 patternContentUnits 的值与 patternUnits 的默认值相匹配。图案内容将继承 0.025px 的描边宽度，它将在 objectBoundingBox 中解析。

❷ 线的坐标同样也被转换为对象边界盒单位。两条线之间相隔 0.025 个单位，第一条线以图案磁贴边的宽度的一半为中心。三条线加上与之大小相等的间隙正好是 0.1 个单位，或者说是边界盒宽度的 10%（图案磁贴的确切宽度）。

当使用小数边界盒单位时，通常最简单的是单独使用数学上的用户单位，不要受到具体单位的干扰，因为它将脱离它在现实世界的意义。当然，你可以使用单位，它们将基于用户单位等比例缩放，stroke-width 的值可以证明这一点。

> 虽然 px 通常可以与 SVG 用户单位交换（在任何坐标系中），但是 Firefox（版本 40）在 stroke-width 定义为 0.025px 时，会四舍五入为 0.03px。但如果定义为 0.025 而没有单位，它是可以正确渲染描边的。

还有另一种代替方案是使用逐渐变小的小数作为坐标。<pattern> 可以使用 viewBox 属性来给每个磁贴定义它自己的坐标系。我们将在第 11 章中探讨 viewBox 图案，以及对象边界盒图案的更多用途，该章的重点是创建一个使用单个磁贴填充整个边界盒的填充内容。

但是，对于重复磁贴你更有可能会使用 userSpaceOnUse 单位。

10.3　布局磁贴

patternUnits 和 patternContentUnits（默认）的值都为 userSpaceOnUse 时，你可以创建一个固定尺寸的磁贴，每个磁贴里面都有一个固定尺寸的图形。这种结果更接近现实世界的磁贴，你可能会使用它们铺地板或墙壁。磁贴不会根据地板区域的大小而改变，只不过铺满空间所需的磁贴数量不同。

例 10-4 改编了例 10-1 中相同的条纹图案，但是这次我们固定条纹的尺寸而去调整磁贴数量。结果如图 10-4 所示。

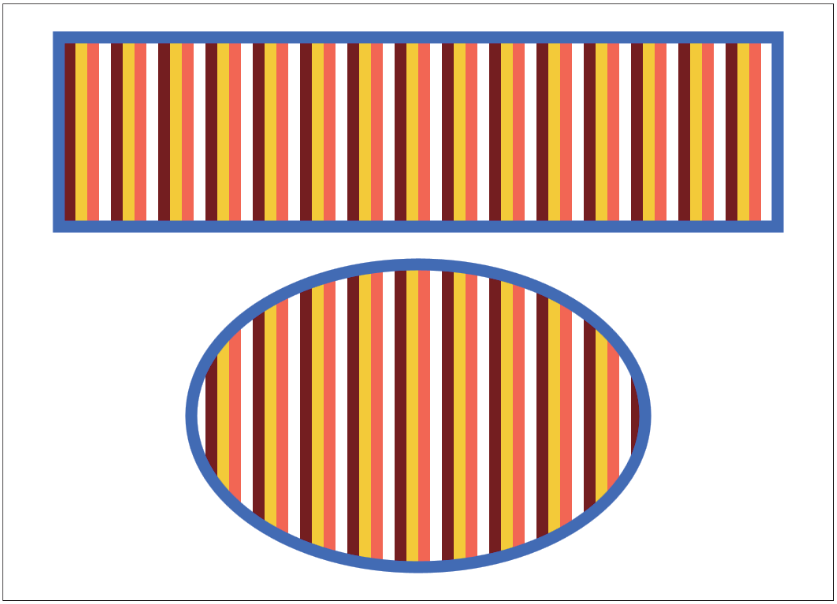


图 10-4：基于用户空间单位的重复条纹图案填充的形状

例 10-4　使用基于用户空间单位的重复条纹图案来填充形状

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="3in" >
    <title xml:lang="en">UserSpace Striped Pattern</title>
    <style type="text/css">
        svg {
            stroke-width: 6px;
        }
    </style>
    <pattern id="stripes3" x="12px" width="24px" height="10px"
             patternUnits="userSpaceOnUse">                       ❶
        <line x1="3" x2="3" y2="10" stroke="maroon" />
        <line x1="9" x2="9" y2="10" stroke="gold" />              ❷
        <line x1="15" x2="15" y2="10" stroke="tomato" />
    </pattern>

    <g stroke="royalBlue" fill="url(#stripes3)">
        <rect x="0.1in" y="0.1in" width="3.8in" height="1in" />
        <ellipse cx="50%" cy="2.1in" rx="1.2in" ry="0.8in" />
    </g>
</svg>
```

❶ 这次明确设置 patternUnits 的值与 patternContentUnits 的默认值相匹配。图案磁贴属性重新使用 px 单位来定义。

❷ 线条不需要拉伸到 SVG 高度的 100%，因为我们知道磁贴的尺寸，可以设置它的确切值。

长方形和椭圆内的条纹不仅尺寸相同，而且完美对齐。用户空间图案磁贴的 x 和 y 偏移是相对于主坐标系的原点来计算的，而不是形状边界盒的角。类似于我们在第 7 章中讨论的用户空间渐变，这可以创建从一个形状到另一个形状一致的渲染流——除非形状被变换。

因为图案磁贴和图案内容基于相同的坐标系，我们可以创建比无限拉伸直线更复杂的图案。在磁贴的中心放置一个圆，调整图标的大小使它恰好贴满磁贴，或者使用 <image> 来填充磁贴。

当你希望图案中相邻的磁贴能够平滑过渡，就好像图 10-4 中垂直的条纹那样，通常最好的做法是使图形稍大于图案磁贴。在一些 SVG 查看器中，你可能会看到例 10-4 中的代码创建的每个图案磁贴都有或深或浅的边缘，这是由于浏览器对形状的像素格进行四舍五入处理导致的。把每个条纹从 y1="-1" 延长至 y2="11"（而非 0 到 10）通常可以解决这个问题，线条超出部分的端点会被裁掉。

你可以利用裁剪效果来创建初看起来不是通过长方形重复磁贴生成的图案。你可以重复形状，重叠磁贴相对的边，这样当磁贴一个挨着一个排列时，形状看起来就是一个个图案循环呈现。

图 10-5 显示了一个重叠鳞片的图案，就好像热带鱼的鱼鳞。虽然最小重复单元是一个交错重叠的圆，但你可以通过绘制从一个鳞片的中心到下一个鳞片的中心然后转向侧边直接向下的正方形来创建一个简单的长方形重复图案，如图下半部分所示。例 10-5 提供了图案的代码，包括重复单元的放大版本。

例 10-5　使用重叠形状创建复杂的图案

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="6.5in" viewBox="0 0 400 650">
    <title xml:lang="en">Fish Scale Pattern</title>
    <style type="text/css"><![CDATA[
        .scale {
            fill: url(#scale-gradient);
            stroke: black;
        }
```

```
    ]]> </style>
    <defs>
        <radialGradient id="scale-gradient">                  ❶
            <stop stop-color="#004000" offset="0"/>
            <stop stop-color="green" offset="0.85"/>
            <stop stop-color="yellow" offset="1"/>
        </radialGradient>
        <pattern id="scales-pattern" width="20" height="20"
                 patternUnits="userSpaceOnUse" >               ❷
            <g id="scales">
                <circle class="scale" cx="0"  cy="19" r="10"/> ❸
                <circle class="scale" cx="20" cy="19" r="10"/>
                <circle class="scale" cx="10" cy="9"  r="10"/> ❹
                <circle class="scale" cx="0"  cy="-1" r="10"/>
                <circle class="scale" cx="20" cy="-1" r="10"/> ❺
            </g>
        </pattern>
    </defs>
    <rect width="400" height="400" fill="url(#scales-pattern)"/> ❻
    <g transform="translate(200,525) scale(5) translate (-10,-10 )">
        <use xlink:href="#scales" />                           ❼
        <rect width="20" height="20" fill="none"
              stroke="deepSkyBlue" stroke-width="0.5"/>        ❽
    </g>
</svg>
```

❶ `<radialGradient>` 用于填充图案中每个鳞片。

❷ 图案本身 `patternUnits` 的值为 `userSpaceOnUse`，意思是图案磁贴是 20px 的正方形，而不管填充形状的尺寸。

❸ 前两个鳞片分别以图案磁贴下边的角为中心。

❹ 下一个鳞片近似放置在磁贴的中心，且在前两个鳞片的上边。然而，为了避免描边被裁减掉，所有的圆都向上移动了一个单位。

❺ 最后两个鳞片以磁贴上边的角为中心。它们会覆盖在中间的鳞片之上，且与磁贴下面一行的鳞片连在一起，且同样分别位于左边和右边。

❻ 图形中上边的长方形显示的是图案连续填充的鳞片。

❼ 图形的下半部分，`<use>` 元素复制了一份五个鳞片组成的组。通过 `<g>` 元素上的 `transform` 属性，鳞片被放大五倍，之后给磁贴添加偏移使它位于可见空间的中心。

❽ 蓝色的 20×20 的长方形，同样绘制在被放大后的坐标系中，它描绘了图案磁贴的边界轮廓。

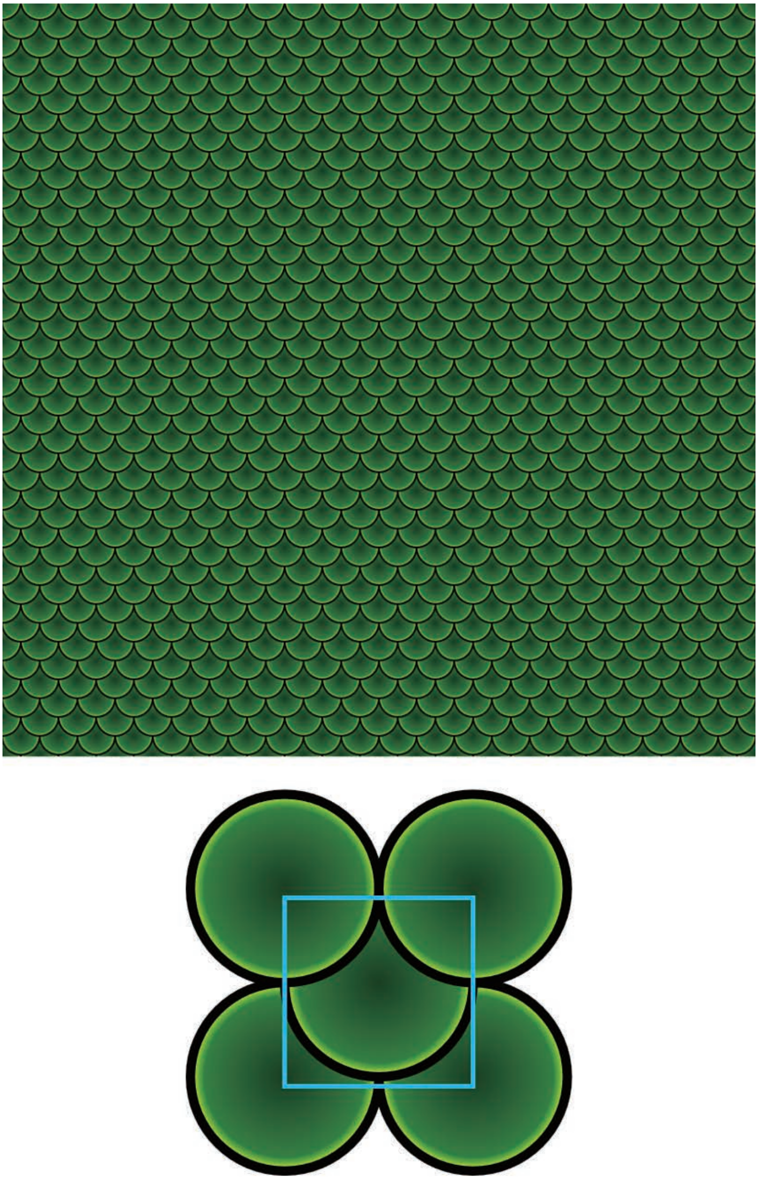


图 10-5：用于填充长方形的鱼鳞（上）以及重复图案磁贴单元（下）

通过这种方式可以构建许多图案，但可能另外需要一些数学知识来计算它的尺寸。你可以标识出重复图案在水平和垂直位移上的确切位置，这有助于描绘图案块最终的外观。它就是你最终图案磁贴的边界。

CSS 与 SVG
重复背景图片

在 CSS 布局中，可以通过背景图片来创建重复图案。因此图案内容可以通过任何有效的 CSS 图片类型来创建：光栅图片，SVG 图片或者 CSS 渐变。默认情况下，图片会在水平和垂直两个方向上重复来创建类似于 SVG 图案的磁贴效果。

起初，CSS 背景图片通常以图片本来的尺寸来绘制。类似于用户空间图案，改变元素背景的尺寸会改变重复单元的数量，而不会缩放它们。对于大型背景图片来说这似乎没有什么用，这时我们更希望它能够缩放到适合元素的大小，对于所有的渐变（以及部分 SVG 图片）也没有什么用，因为它们本身没有尺寸。CSS Backgrounds and Borders Module Level 3 中引入了 `background-size` 属性，它允许每个背景图片可以缩放到一个固定尺寸或者元素的百分比。

相比于 SVG 图案，CSS 背景图片也有一些优点。

- 使用 `background-repeat` 属性可以让背景只在一个方向上重复或根本就不重复。
- `background-size` 属性还接受一个 `auto` 值来使图案磁贴的宽或高缩放到匹配其他值或内容本身的宽高比。
- 背景可以有多层，每层都可以拥有自己的尺寸和重复设置。

正如我们多次提到的那样，分层渲染服务填充将在 SVG 2 中加入。在第 11 章中，我们将介绍如何通过拼合图案来模拟这种效果。重复的控制可以通过创建比对象边界盒更宽或更高的图案磁贴来模拟。

使用 CSS 背景来创建图案效果的主要不足之处是图案内容必须是一个单独的图片文件（或编码为 data URI），除非可以使用渐变来表示。虽然渐变可以用于创建块和条纹，但在一些浏览器中其渲染质量明显不如 SVG 形状。

10.4　变换磁贴

在上一节中，我们介绍了如何通过在每一个磁贴内包含复杂的重复单元来创建非矩形图案。对于某些几何图案，你可以通过使用坐标系变换来精简

代码，同时实现相同的效果。

patternTransform 属性允许旋转、倾斜、缩放或移动图案。变换不仅作用在图案的内容，而且作用在整个图案磁贴，作用在磁贴端到端平铺的重复图案。

> 与 gradientTransform 类似，CSS 变换模块重新把 pattern-Transform 定义为一个表现属性，与 transform 样式属性意思相同——虽然现在任何浏览器都不支持。

图案变换使得创建对角线以及其他角度的图案变得很容易，不需要通过三角形学来计算出重复的线之间精确的水平或垂直距离。例 10-6 中使用 45 度的旋转来创建斜的细条纹和网格图案。本例中还演示了使用图案作为另一图案的模板的多种方式。最终结果如图 10-6 所示。

例 10-6　使用 patternTransform 创建对角图案

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="4in" viewBox="0 0 400 400">
    <title xml:lang="en">Pinstripe Patterns</title>
    <defs fill="#444" stroke="lightSkyBlue">                    ❶
        <pattern id="pinstripe" patternUnits="userSpaceOnUse"
                 width="30" height="30">                        ❷
            <rect id="r" width="30" height="30"
                 stroke="none" />
            <line id="l" x1="15" y="0" x2="15" y2="30"
                 fill="none" />
        </pattern>
        <pattern id="diagonals" xlink:href="#pinstripe"
                 patternTransform="rotate(45)" />               ❸
        <pattern id="grid" xlink:href="#pinstripe">
            <use xlink:href="#r" />                             ❹
            <use xlink:href="#l" />
            <use xlink:href="#l" transform="rotate(90, 15, 15)"/>
        </pattern>
        <pattern id="diagonal-grid" xlink:href="#grid"
                 patternTransform="rotate(45)" />               ❺
    </defs>

    <rect width="200" height="200"
          fill="url(#pinstripe)" />
    <rect width="200" height="200" x="200"
          fill="url(#diagonals)" />
    <rect width="200" height="200" y="200"
          fill="url(#diagonal-grid)" />
    <rect width="200" height="200" x="200" y="200"
          fill="url(#grid)" />
</svg>
```

❶ 四个图案使用相同的配色方案，通过这里的 <defs> 元素设置。和形状一样，图案和其他渲染服务会从它们所处的环境继承样式。

❷ 核心的 pinstripe 图案磁贴由纯色填充的正方形和一条沿着中心垂直向下的线组成。磁贴相对于用户空间坐标系的尺寸是固定的。

❸ diagonals 图案复制了第一个细条纹图案，并把它旋转 45 度创建了对角线。

❹ grid 图案依然使用 pinstripe 图案元素作为模板，并继承图案元素上的属性，但它通过复制一次纯色长方形和两次线条，并把其中一条旋转 90 度来替换图案的内容。与渐变一样，只要 <pattern> 有子内容，它就会替换模板的所有内容。

❺ 最后一个图案复制并旋转了 grid 图案。

例 10-6 还强调了图案与其内容是如何根据它们在文档树的位置来继承样式的。图案不会继承使用图案填充的元素上的样式。

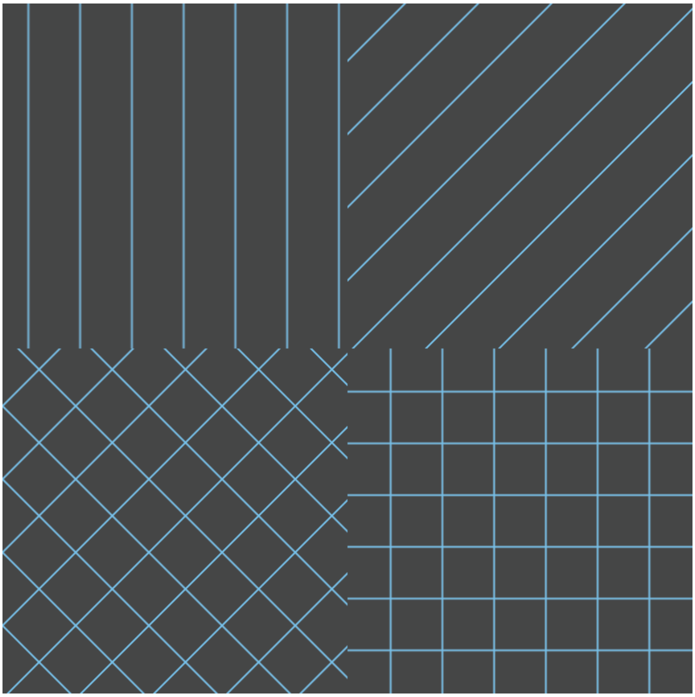


图 10-6：旋转变换前后的细条纹图案

> xlink:href 图案模板和图案样式继承的结合意味着你可以通过改变继承的样式来创建颜色不同的图案，类似于你可以通过设置 <use> 元素的样式来创建颜色不同的图标。不幸的是，SVG 规范的起草者没有考虑到这种可能性，且任何已经测试的浏览器都没有实现这种方式。
>
> 然而，规范中也没有明确说明复制图案内容不应该从新的图案中继承，所以在未来这可能会有所改变。目前，在原始和复制的图案元素中，所有继承样式都是相同的。

这种对角线效果同样可以使用对象边界盒图案磁贴创建，但在 7.4 节中的经验教训依然适用。对象边界盒单元会添加自己的变换，这会扭曲旋转的角度。旋转 45 度将会创建沿着边界盒对角线的线，而不论该 45 度是不是绝对意义上的 45 度。

聚焦未来

简写的 hatch 图案

SVG 2 引入了一个新的渲染服务元素，它将大大简化像例 10-6 中那样的条纹图案的创建。我们把它称为 hatches，它们也用于消除图案磁贴边缘的不连续（四舍五入的误差所致）。与磁贴在两个方向上重复不同，它本身是由条纹组成的。条纹将会像图案那样在一个方向上平铺，但它们是由一条连续的路径组成，且在另一个方向上无限长。每条路径都会有重复的部分，但它会作为一个单独的元素平滑地渲染。

hatch 图案通过 <hatch> 元素来定义，它有以下属性：

- x 和 y（定义第一个条纹的偏移位置）
- pitch（条纹之间的间隔）
- rotate（条纹的角度）
- hatchUnits 和 hatchContentUnits（值为 userSpaceOnUse 或 objectBoundingBox，默认情况下与图案相同）
- transform（等同于 patternTransform）
- xlink:href（引用另一个 <hatch> 元素来作为模板）

hatch 元素包含一个 <hatchPath> 元素，每一个该元素都定义了一条线或路径。默认情况下，路径是一条垂直线（取决于父级 <hatch> 的变换）。你还可以使用 d 属性来提供路径定义的一个片段：你指定的路径将会无限连接

在一起。因此如下代码将会创建无穷的波浪线：

```
<hatch hatchUnits="userSpaceOnUse" pitch="6">
    <hatchPath stroke-width="1"
               d="c 0,4 8,6 8,10 8,14 0,16 0,20"/>
</hatch>
```

无法给 `hatch` 图案创建背景颜色，因为有了新的分层填充的语法，背景颜色可以在使用 `hatch` 的同时定义。

通过变换，你可以创建平行四边形或菱形图案磁贴来布局，而不局限于简单的长方形。对于大多数可预期的结果，这类图案通常使用 `userSpaceOnUse` 效果最好。对象边界盒单位的不均匀缩放可能无法与 `patternTransform` 值完美搭配。

图 10-7 显示了两种可以通过变换矩形磁贴来创建的几何图案。代码如例 10-7 所示。

例 10-7　使用 `patternTransform` 创建三角形或菱形图案

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="6.5in" viewBox="0 0 400 650">
    <title xml:lang="en">Transformed Patterns</title>
    <pattern id="triangles" patternUnits="userSpaceOnUse"
             width="20" height="17.32"
             patternTransform="skewX(30)">                       ❶
        <rect width="30" height="20" fill="lightGreen" />
        <polygon points="0,0 20,0 0,17.32" fill="forestgreen" /> ❷
    </pattern>
    <pattern id="argyle" patternUnits="userSpaceOnUse"
             width="20" height="20"
             patternTransform="scale(2,4) rotate(45)">           ❸
        <rect fill="mediumPurple" width="20" height="20"/>
        <rect fill="indigo" width="10" height="10"/>             ❹
        <rect fill="navy" width="10" height="10"
              x="10" y="10"/>
        <path stroke="lavender" stroke-width="0.25" fill="none"
              d="M0,5 L20,5 M5,0 L5,20
                M0,15 L20,15 M15,0 L15,20" />                    ❺
    </pattern>


    <rect width="400" height="325" fill="url(#triangles)" />
    <rect width="400" height="325" y="325"
          fill="url(#argyle)" />
</svg>
```

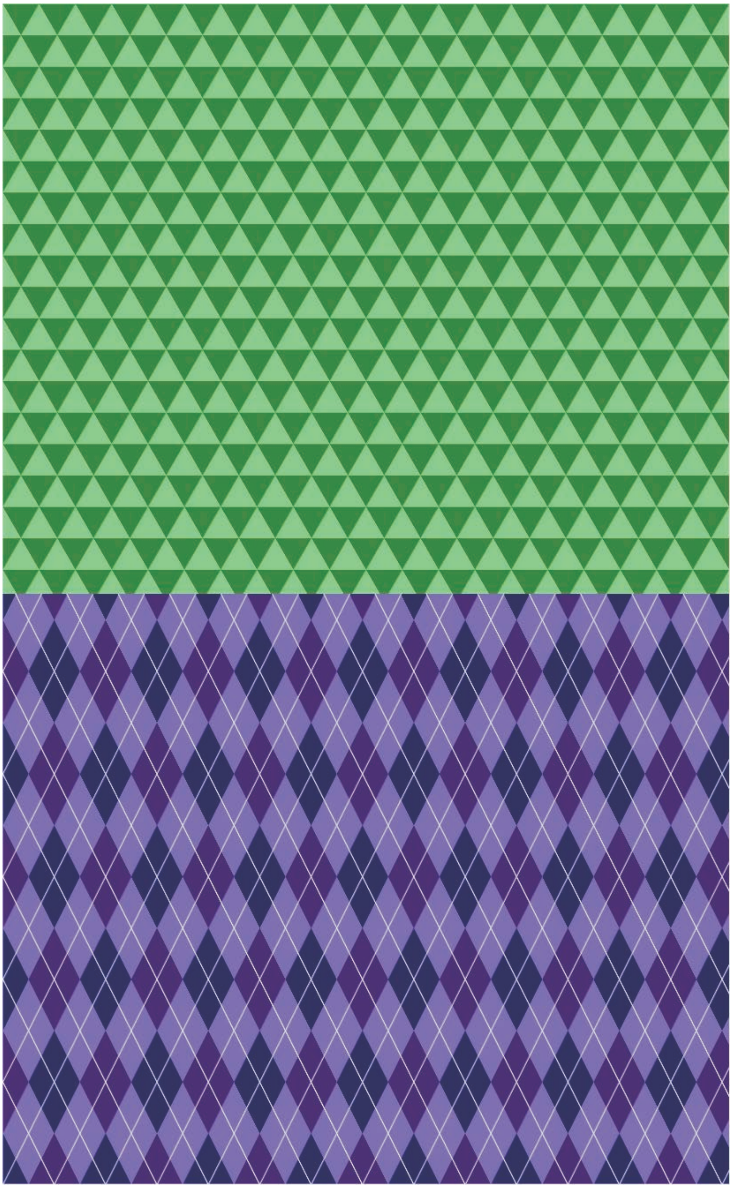


图 10-7：使用变换的图案磁贴创建的三角形和多色菱形图案

❶ 第一个图案使用 30 度的倾斜来把图案磁贴转换为平行四边形。每个磁贴的高度被调整为等边三角形的高，三角形的底等于图案磁贴的宽度。

❷ 该步骤的结果是，填充未变换的磁贴左上角的 `<polygon>` 三角形在倾斜 30 度后变为一个完美的等边三角形。

❸ 第二个图案使用不均匀的缩放以及 45 度的旋转来创建一个菱形图案磁贴。

❹ 方格图案是通过使用两个位于相对角且宽和高都为磁贴一半的长方形来创建。该方格会与图案磁贴一起变换来创建方格菱形。

❺ 菱形图案的最后一个元素是绘制四条单独的直线的 `<path>`，它们用于平分每个菱形方格。

例 10-7 中的两个图案都在垂直和水平上重复，并且都可以不使用变换来创建。但这将导致图案内容更加复杂。更多的坐标需要进行三角形计算，并且很可能会由于四舍五入导致相邻的磁贴之间出现不连续。

CSS 与 SVG
复杂的重复图案

CSS 背景图片不能在它们的宿主形状内单独变换。图 10-6 和图 10-7 这样的图案有两种创建方法。

- 使用伪元素来包裹重复的背景，设置它的 `z-index` 来使它位于主内容下面，然后给它添加 `transform` 属性来应用旋转、倾斜和缩放。确保伪元素的 `display` 为 `block` 且大于它的父级元素（这样变换的时候不会漏出空白点），还要确保父级元素的 `overflow` 设置为隐藏。
- 绘制一个很大的背景，计算水平和垂直方向上确切的重复数量，然后在每个背景磁贴内引入变换效果。这种方式与例 10-5 中创建重叠的鱼鳞效果类似。

第 11 章

完美的图片图案

pattern 引出了我们在第 10 章中探讨的重复设计。然而，SVG 中的 <pattern> 元素比这更加灵活。通过定义适合对象边界盒的图案磁贴，你可以创建不通过重复而填充整个形状的图案。

那么为什么要这样做呢？一是能够同时布局多个渲染服务。二是使用图片填充形状，就像 CSS 背景图片那样。

之前我们已经简要地提到，SVG 2 允许在一个单独的填充声明中使用多层渲染服务，且可以直接使用图片来作为填充值，这似乎使得这种类型的图案显得很多余。然而，就算这些属性都可用，这项技术在创建多数效果时也必不可少。

本章还讨论了在 <pattern> 元素上使用 viewBox 和 preserveAspectRatio 属性。这些属性在全图像填充时尤其重要，它们也可以与重复图案一起使用。

11.1　层次感

在第 9 章最后的例 9-8 中，我们分别给三个独立的矩形使用不同的渐变填充，创建了聚光灯照射在舞台上的效果。这种方式对于单个形状来说很有用，但是如果你想在多个形状中使用相同的分层效果，这种方式就不太适合了。此时就应该使用图案了。一个图案可以包含所有的层，把它们变成一个单独的渲染服务来供其他形状使用。

例 11-1 中把三个渐变结合在一个 <pattern> 中来使一个圆和一个椭圆看起来像颜色亮度不均匀的球体。相同的图案也可以用来填充文本标题，如图 11-1 所示。

例 11-1　使用非重复图案的分层渐变

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4in" height="6.5in" viewBox="0 0 400 650">
    <title xml:lang="en">Multiple gradients</title>
    <defs>
        <linearGradient id="red-blue" y2="1">                    ❶
            <stop stop-color="red" offset="0"/>
            <stop stop-color="blue" offset="1"/>
        </linearGradient>
        <linearGradient id="yellow-violet" y1="1" y2="0">  ❷
            <stop stop-color="yellow"
                  stop-opacity="0.9" offset="0.1"/>
            <stop stop-color="darkMagenta"
                  stop-opacity="0" offset="0.5"/>                ❸
            <stop stop-color="violet"
                  stop-opacity="0.9" offset="0.9"/>
        </linearGradient>
        <radialGradient id="flare" fx="0.2" fy="0.2"
                        stop-color="white" >                     ❹
            <stop stop-color="inherit"
                  stop-opacity="0" offset="0.75"/>
            <stop stop-color="inherit"
                  stop-opacity="0.05" offset="0.85"/>
            <stop stop-color="inherit"
                  stop-opacity="0.2" offset="1"/>
        </radialGradient>
        <pattern id="gradient-pattern" width="1" height="1"
                 patternContentUnits="objectBoundingBox" >  ❺
            <rect width="1" height="1" fill="url(#red-blue)"/>
            <rect width="1" height="1" fill="url(#yellow-violet)"/>
            <rect width="1" height="1" fill="url(#flare)"/> ❻
        </pattern>
    </defs>
    <g fill="url(#gradient-pattern)">                           ❼
        <circle cx="200" cy="180" r="180" />
        <ellipse cx="110" cy="500" rx="110" ry="145" />
        <text x="400" y="525" text-anchor="end"
              font-size="100px" font-family="serif"
              stroke="indigo">Layers</text>
    </g>
</svg>
```

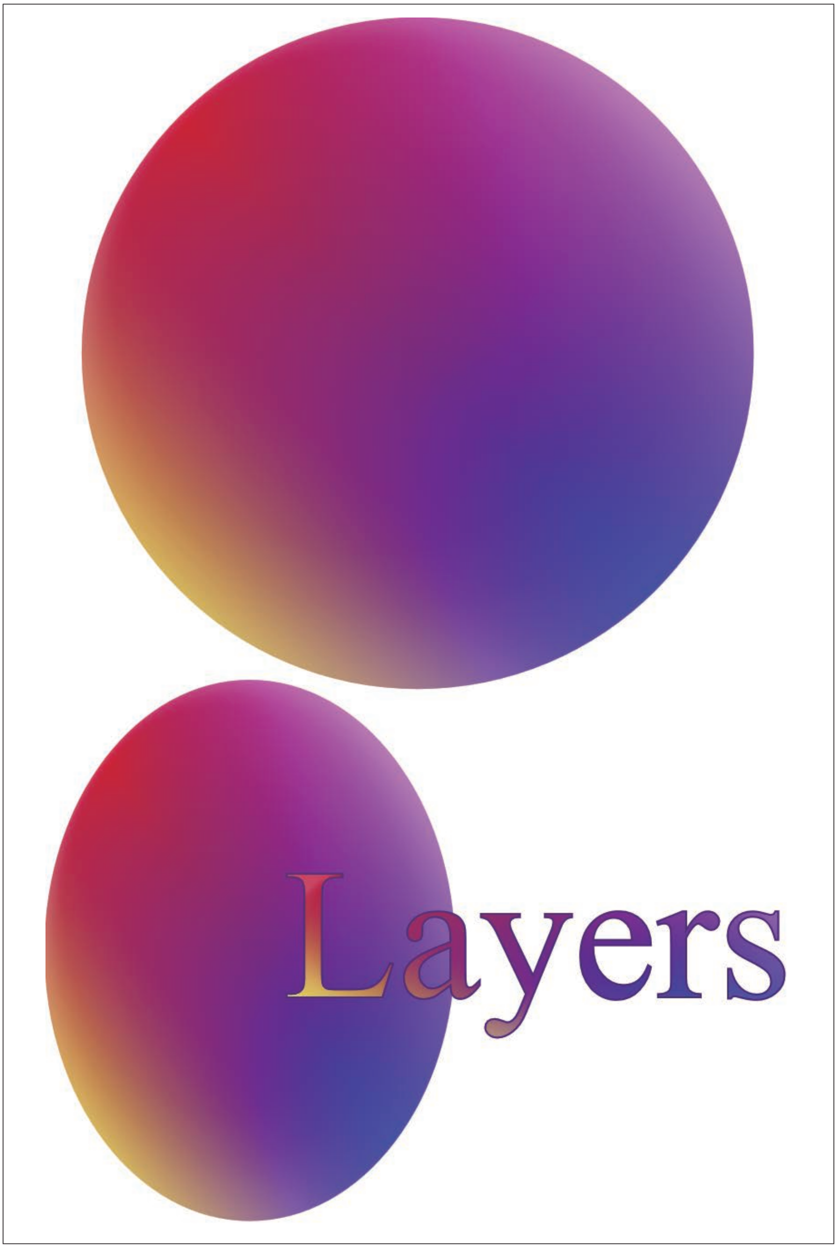


图 11-1：分层渐变图案填充的形状和文本

❶ red-blue 渐变从边界盒的左上角向右下角（记住，x2 默认是 100%）延伸。它使用的是纯色且位于最底层。

❷ yellow-violet 渐变从边界盒的左下角向右上角延伸。

❸ 该渐变局部透明，在中点位置完全透明（stop-opacity="0"）以允许其他层显示出来。透明的结点依然需要一个 stop-color，否则它会默认为黑色且两边的颜色将会是暗灰色的色调。

❹ 白色的径向渐变几乎是透明的，只有在边缘是 20% 的不透明。为了方便编辑或添加动画，单一颜色定义在渐变元素上，然后在每个结点上都明确指定继承父级。不幸之处是 stop-color 默认不会继承。

❺ <pattern> 中图案磁贴（默认）和内容（明确指定）使用的都是 objectBoundingBox 单位。磁贴的宽和高都设置为不重复填满整个盒子。

❻ 渐变层通过填充 <rect> 元素来绘制，它也会被缩放到填充整个边界盒，宽和高都是 1 个单位。

❼ 组元素上使用 fill 表现属性来设置使用分层图案填充，形状和文本将会继承该值。然而，每个元素都独立继承填充属性，且使用它自己的边界盒来定义渐变层。

图 11-1 中的图案，以及组成它的渐变，会根据每个形状（或文本元素）对象边界盒的尺寸来拉伸。对于抽象的渐变，这没有什么问题。但对于其他图案内容，这种扭曲是不可接受的。

避免扭曲的一种方式是给图案内容使用 userSpaceOnUse 单位。然而，这样的话图案就不会缩放到适合形状的大小。此外，坐标的位置是相对于图形的整体空间，而不是对象本身。如果移动对象，它的坐标就会改变，图案会保持相对于背景固定。

通过相对于坐标系中一个固定的点来定义形状，然后在 <use> 元素上使用变换或 x 和 y 属性（和变换有相同效果）来把它移动到合适的位置，我们可以处理定位的问题。要想解决缩放的问题，只能使用 <symbol> 元素或嵌套的 SVG 来为每个要填充其整个宽和高的形状创建一个单独的坐标系。当然，这一切听起来都会增加许多额外的工作和复杂性。

幸运的是，还有一个选项可以控制 <pattern> 内容的缩放：定义一个 viewBox 并使用 preserveAspectRatio 来保证它不会被扭曲。

11.2 保持原始图案

如果没有 viewBox 属性，你的 SVG 就不能很好地工作，尤其是在 Web 上。把它设置在你的根 <svg> 元素上，它会建立基本的坐标系以及宽高比，以允许你的图形可以缩放到设置的任何区域。它也可以用在嵌套的 <svg> 元素上和重复使用的 symbol 中来创建局部缩放的效果。

viewBox 可以定义四个数字：前两个定义的是你要在图形中包含的最小点的 (x,y) 坐标，而第三个和第四个数字分别表示的是图形包含的宽和高的单位数。通常前两个数字都是 0（所以原点在左上角且图形中所有的坐标值都是正的）或负值 (用于把坐标原点放置在图形中心)。

preserveAspectRatio 属性很容易被我们忽视，因为它的默认值往往就是你需要的值。它可以设置为以下三种缩放模式的一种。

- meet 在不扭曲的情况下，缩放你整个 viewBox 区域到正好全部显示在（紧贴边缘）可见空间内。
- slice 放大 viewBox 到覆盖整个可见空间（多余部分会被裁减掉）
- none 按需要拉伸或压缩 viewBox 到正好在两个方向上填充可见空间。

对于 meet 和 slice，必须还要定义一个类似于 xMinYMax 的基准值来控制 viewBox 区域内哪个点与绘制矩形对等的点相对准。

在 <pattern> 元素中，viewBox 属性会覆盖 patternContentUnits 的设置并给你的图案创建自己的坐标系。就任何 preserveAspectRatio 属性值而言，viewBox 创建的坐标系都将缩放到适合图案磁贴。



在 Firefox 40（2015 年中发布）之前，viewBox 以及任何 preserveAspectRatio 值是在转换为对象边界盒之后应用的（忽略 patternContentUnits 的值）。这会导致非正方形边界盒中的图案被扭曲，即使图案的宽高比与形状完全匹配。

因为 meet 会维持图案的宽高比来缩放到适合可见空间的整个宽和高，这可能会导致图案周围的内容出现空白。这包括 xMidYMid meet 值应用在定义 viewBox 而没有定义 preserveAspectRatio 的元素上时。

有一种方式是有意地在你的图案内绘制一个比 viewBox 更大的背景元素。该背景会被裁剪到图案磁贴的大小，且 viewBox 将会缩放到适合它的大小。例 11-2 中使用这种方式创建了一个填补的圆形渐变并保持了它的本来形状，

而忽略要填充形状的宽高比。保持原始形状的渐变绘制在一个正方形形状内（也可以使用 circle）。该正方形会填充整个 viewBox，但 viewBox 通常不会填充整个图案磁贴。

图 11-2 中显示了渐变图案的结果，并填充了一个长方形。

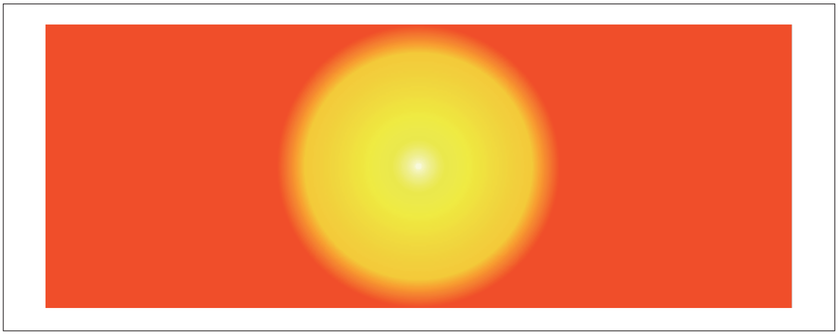


图 11-2：通过固定宽高比的图案而放置在长方形盒子内的圆形渐变

例 11-2　使用固定宽高比的图案创建一个永远都是圆形的渐变

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="100%" height="100%">                          ❶
    <title xml:lang="en">Always-Circular Gradient</title>
    <defs>
        <radialGradient id="radial">                      ❷
            <stop stop-color="lightYellow" offset="0"/>
            <stop stop-color="yellow" offset="0.2"/>
            <stop stop-color="gold" offset="0.8"/>
            <stop stop-color="orangeRed" offset="1"/>
        </radialGradient>
        <pattern id="circular-gradient" width="1" height="1"
                 viewBox="0 0 1 1">                       ❸
            <rect width="5" height="5" x="-2" y="-2"
                  fill="orangeRed"/>                      ❹
            <rect width="1" height="1"
                  fill="url(#radial)"/>                   ❺
        </pattern>
    </defs>
    <rect height="100%" width="100%" fill="url(#circular-
gradient)"/>
</svg>
```

❶ SVG（以及它包含的长方形）占据可见区域的 100%，所以你可以通过缩放浏览器窗口来测试不同的宽高比。

❷ 径向渐变使用默认的位置和尺寸，所以它将放置在要填充形状的中心，并缩放到其边界盒大小。

❸ 图案也会填满整个边界盒（宽和高为 1，且 `patternUnits` 使用默认值）。`viewBox` 定义了内容单位，创建了一个 1:1 的宽高比，而忽略边界盒的尺寸。

❹ 背景矩形绘制为 `viewBox` 创建的正方形区域的 5 倍大小，并以它为中心。背景使用最后一个渐变结点（填补）中相同的颜色填充。

❺ 渐变本身绘制在一个宽和高正好适合 `viewBox` 的正方形内。

值得注意的重要一点是，图案内的形状是通过缩放的用户单位而不是百分比定义的。虽然 `viewBox` 创建了新的坐标系原点和新的缩放，但它没有创建一个定义百分比长度的新视口。在图案内，百分比长度会基于某一单位按比例缩放，而忽略任何 `viewBox` 的值。

> 正如对象边界盒单位中提到的，这通常并没什么用且不可预见，所以应避免在除 `userSpaceOnUse` 外的图案中使用百分比。

使用 `viewBox` 而非对象边界盒单位还可以消除图案磁贴缩放和图案内容缩放之间的相互依赖。图案上的 `viewBox` 通常会缩放到适合每个单独的图案磁贴，而不是整体的边界盒。图 11-3 显示了缩小每个图案磁贴到盒子宽度的 10%、高度的 25% 时图案的样子，使用的是如下属性：

```
<pattern id="circular-gradient" width="0.1" height="0.25"
         viewBox="0 0 1 1">
```

图案内容中 `<rect>` 元素上的宽和高都没有改变。

例 11-2 中使用超大背景的方式并不理想。如果图案磁贴的宽高比比 5:1 更极端，背景就不能填满整个图案。如果使用图案覆盖整个磁贴比让整个图案内容可见更加重要，你可以给 `preserveAspectRatio` 使用 `slice`。它可以保证图案内容总是完全覆盖磁贴。

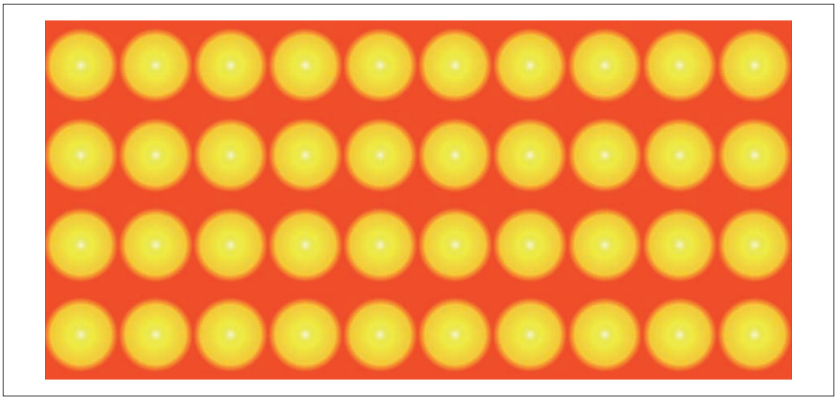


图 11-3：重复的圆形渐变图案

11.3　SVG样式的背景图片

当使用 slice 方式来保持宽高比，且是单磁贴边界盒图案时，它的效果与 CSS 中设置背景图片 background-size: cover 时很相似。单个图形会填充整个形状。

例 11-3 使用一个图案来包含一个阳光和天空的 SVG 图形，它可能会被用于幻灯片或海报的背景。slice 效果通过一系列填充不同宽高比的矩形的图案来展示，结果如图 11-4 所示。

例 11-3　重复使用一个 SVG 图形来填充非重复图形

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4in" height="6.5in" viewBox="0 0 400 650">
    <title xml:lang="en">Sliced Image Pattern</title>
    <defs>
        <linearGradient id="sky" x2="0" y2="1">          ❶
            <stop stop-color="lightSkyBlue" offset="0"/>
            <stop stop-color="deepSkyBlue" offset="1"/>
        </linearGradient>
        <radialGradient id="sunlight" cx="0" cy="0" >
            <stop stop-color="yellow"
                  stop-opacity="0.9" offset="0.2"/>
            <stop stop-color="lightYellow"
                  stop-opacity="0" offset="1"/>
        </radialGradient>
        <radialGradient id="cloud" fx="0.5" fy="0.15" r="0.6">
            <stop stop-color="oldLace" offset="0.75"/>
```

```
            <stop stop-color="lightGray" offset="0.9"/>
            <stop stop-color="darkGray" offset="1"/>
        </radialGradient>
        <pattern id="sky-pattern" width="1" height="1"
                 viewBox="0 0 100 50"
                 preserveAspectRatio="xMinYMin slice" >    ❷
            <rect width="100" height="50" fill="url(#sky)"/>
            <g fill="url(#cloud)">                           ❸
                <g>
                    <circle cx="10" cy="42" r="5" />
                    <circle cx="6" cy="42" r="3" />
                    <circle cx="16" cy="43" r="3" />
                    <circle cx="14" cy="41" r="4" />
                </g>
                <g>
                    <circle cx="20" cy="22" r="7" />
                    <circle cx="50" cy="22" r="10" />
                    <circle cx="40" cy="18" r="7" />
                    <circle cx="45" cy="25" r="9" />
                    <circle cx="30" cy="25" r="12" />
                </g>
                <g>
                    <circle cx="72" cy="39" r="5" />
                    <circle cx="77" cy="40" r="3" />
                    <circle cx="83" cy="41" r="4" />
                    <circle cx="80" cy="36" r="5" />
                    <circle cx="76" cy="35" r="3" />
                    <circle cx="86" cy="39" r="3" />
                </g>
            </g>
            <rect width="50" height="50" fill="url(#sunlight)"/>
        </pattern>
    </defs>
    <g fill="url(#sky-pattern)">                             ❹
        <rect width="400" height="175" />
        <text x="200" y="280" textLength="390"
            text-anchor="middle" font-family="sans-serif"
            font-size="124px" font-weight="bold"
            stroke-width="2" stroke="deepSkyBlue"
            >Clouds</text>
        <rect y="290" width="250" height="250" />
        <rect x="270" y="290"
              width="130" height="250" />
        <rect x="0" y="550"
              width="400" height="100" />
    </g>
</svg>
```

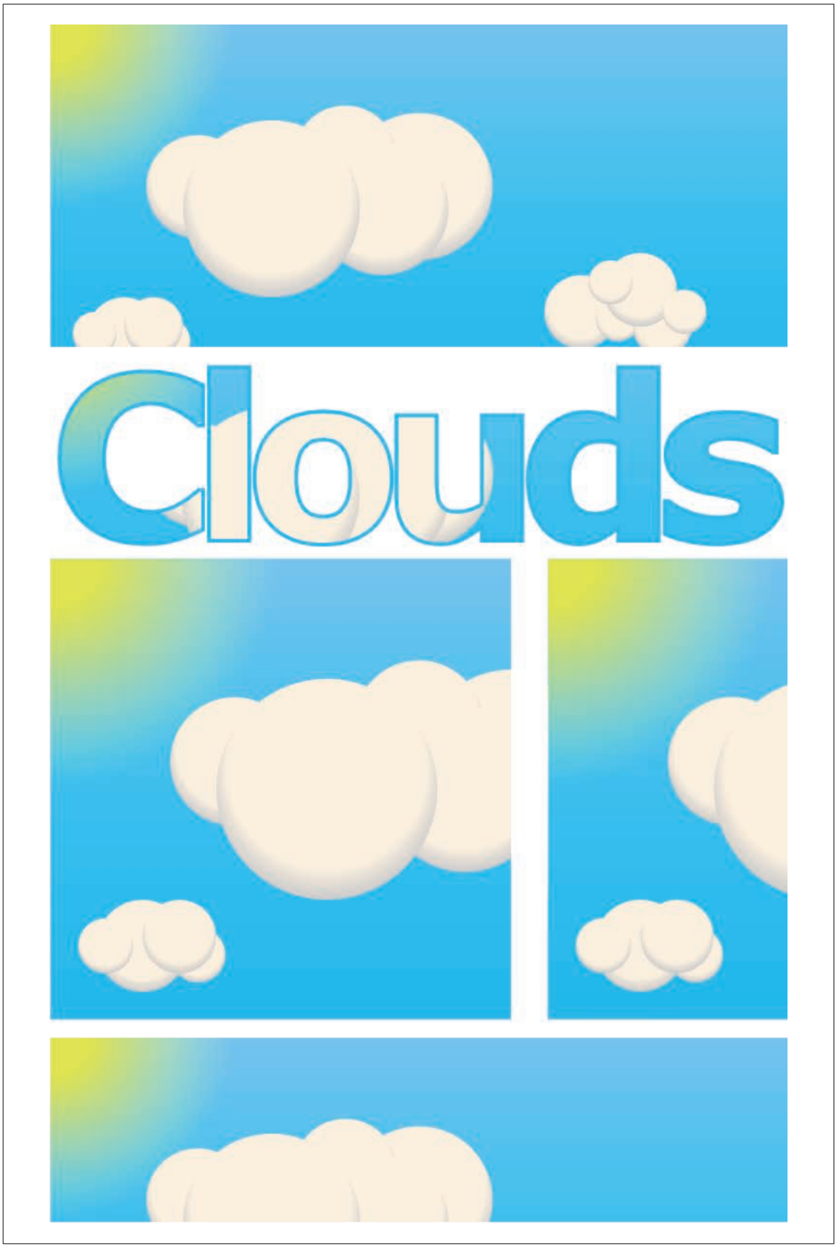


图 11-4：使用相同的图形背景填充不同宽高比的长方形和文本

❶ 该代码包含了许多渐变的定义，它们用在图案内的图形。

❷ 图案本身有一个通过 viewBox 定义的 2:1 (100 × 50) 的宽高比。patternUnits 和 patternContentUnits 属性都不是必需的：patternUnits 属性使用默认值，且宽和高都填充整个边界盒。再次说明，内容单位是不需要的，因为它们会被 viewBox 替换。最后通过 preserveAspectRatio 属性设置 slice 缩放和对齐方式。

❸ 图案内容包括一个蓝色渐变填充的长方形，一组通过渐变填充的圆来创建的云朵，以及最后通过金色到透明的径向渐变形成的太阳填充的长方形。

❹ <rect> 元素和文本都是真实绘制的图形，它们会从组元素上继承填充值。与往常一样，每个填充层的边界盒是基于每个元素计算的，而不是基于组。

图案除了 slice 缩放模式，还使用了 xMinYMin 对齐选项，所以图形的左上角始终包含在边界盒内（虽然形状本身不一定），如填充文本时所示。

聚焦未来
使用图片文件填充 SVG 形状和文本

在第 6 章曾简单提到，SVG 2 中将允许 CSS 图片类型直接作为 SVG 形状和文本的填充值。这包括 CSS 渐变函数，也包括通过 URL 引用单独的 SVG 或光栅图片文件。

例如，你可以使用复用的背景创建一个单独的图片文件，然后将如下规则应用在形状上：

```
.slide {
    fill: url(clouds.svg);
}
```

声明每个填充层尺寸和位置的确切语法在编写本书时还没有敲定，但是语法和使用 CSS 背景图片很相似。

虽然现在还不能使用单独的图片文件代替 SVG 渲染服务，但可以在渲染服务内通过 SVG<image> 元素使用图片文件。

例 11-4 创建了一个包括摄影照片（NASA 提供的地球的合成图片）的图案。图片被放大来填充图案磁贴，多余部分被裁剪掉。图 11-5 显示了最终效果。

例 11-4　在图案内使用摄影照片来填充文本

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="400" height="250" viewBox="0 0 800 500">
    <title xml:lang="en">Earth in Space</title>
    <style type="text/css">
        .earth {
            font-family: sans-serif;
            font-weight: bold;
            font-size: 148pt;

            text-anchor: middle;
            fill: url('#photoFill');
            stroke: #205334;
            stroke-width: 3;
            text-decoration: overline underline;
            text-shadow: white 0 0 8px ;                ❶
        }
        .background {
            fill: url(#background);
        }
    </style>
    <defs>
        <pattern id="photoFill" width="1" height="1"
                 viewBox="0 0 1 1"
                 preserveAspectRatio="xMidYMid slice">      ❷
            <image x="-0.1" y="-0.1" width="1.2" height="1.2"
                   xlink:href="globe_west_1024.jpg" />      ❸
        </pattern>
        <linearGradient id="background"
                        gradientTransform="rotate(90)">     ❹
            <stop stop-color="black" offset="0"/>
            <stop stop-color="black" offset="0.5"/>
            <stop stop-color="navy" offset="0.80"/>
            <stop stop-color="blue" offset="0.9"/>
            <stop stop-color="lightBlue" offset="0.95"/>
            <stop stop-color="green" offset="0.95"/>
            <stop stop-color="brown" offset="1.2"/>
        </linearGradient>
    </defs>

    <rect width="100%" height="100%" class="background"/>
    <text class="earth" x="400" y="60%">EARTH</text>       ❺

    <metadata>                                              ❻
Image Source: http://visibleearth.nasa.gov/view.php?id=57723
    </metadata>
</svg>
```

❶ 字体样式和 SVG 样式都是通过 CSS 类来设置的。text-shadow 在支持该属性的浏览器中给字母周围稍微添加了一些光芒，同时声明 underline 和 overline 是为了添加一个边框效果。

❷ height 和 width 属性创建了一个适合边界盒的单个图案磁贴。viewBox 创建了一个正方形的宽高比，根据 preserveAspectRatio 的值，它将会扩展到覆盖整个磁贴且多余的部分会被裁减掉。

❸ 图片本身绘制得比 viewBox 稍大一些，这样是为了不包含地球周围的黑色空间。

❹ <linearGradient> 提供了一个类似地平线的背景效果。

❺ 文本被放置在 SVGviewBox 的中间，它的尺寸、字体和 text-anchor 都是通过 CSS 类来设置的。

❻ <metadata> 提供了该文件的一些额外信息，这既不是图片的一部分，也不是它的替换文本。



图 11-5：使用裁剪的图案填充的文本

该例子演示了 text-decoration 标记 (underlines、overlines 以及 strike-throughs) 将被视为文本内容的一部分，且都会像字母那样使用相同的样式填充和描边。另一个具体的文本渲染效果是使用 text-shadow 来提高字母周围的对比度。

> 虽然大多数浏览器支持给 CSS 布局的文本添加 `text-shadow`，但 IE 以及许多其他基于 SVG 1.1 规范的工具不支持给 SVG 添加该效果。即使支持的 Web 浏览器中也有一些 bug：Chrome 在文本被缩放时不会缩放它的阴影；Firefox 在文本使用渲染服务填充文本时不会绘制阴影；Safari 在使用渲染服务时，会在每个 em-box 周围创建阴影，而不是沿着字母的形状创建阴影。
>
> 换句话说，只在不添加装饰的文本上使用 `text-shadow`。如果你要使用它，它必须使用 CSS 来定义，而不能通过表现属性定义。

例 11-4 中使用的图片资源如图 11-6 所示。它是一个正方形，宽高比和 `<image>` 元素上的 `height` 和 `width` 属性正好匹配。如果不匹配，默认会缩小来适合 `<image>`，这会破坏图案的 `slice` 效果。对于不同的图片，你可能需要改变图片元素的尺寸以及 `viewBox` 的宽高比，这样图片才可以完全填充。



图 11-6：用于填充 SVG 文本的图片（照片来自美国宇航局戈达德太空飞行中心 Reto Stöckli，由 Robert Simmon 对效果进行增强处理）（http://visibleearth.nasa.gov/view.php?id=57723）

如果不知道要使用的图片的宽高比，你可以在 `<image>` 元素本身上定义一个值为 `slice` 的 `preserveAspectRatio` 属性。两个 `slice` 的设置可能会裁减掉比我们需要的更多的内容，但这也比显示空白更加友好。

你可以把例 11-4 中的效果描述为根据文本裁剪图片。实际上，你可以通过把文本包含在 `<clipPath>` 元素内，然后在 `<image>` 元素的 `clip-path` 属性中引用它来创建非常相似的效果（除去描边和阴影）。这种情况下，`<image>` 将是显示在屏幕上的元素，文本成为图形效果的一部分。通过把图片填充到 `<text>` 元素内，文本可以保持可选择和可访问。

聚焦未来

渲染文本修饰

CSS Text Decoration Module Level 3 中扩展了 `text-decoration` 属性，使它成为一系列子属性的简写形式。扩展的语法不仅允许你设置线条的类型（下划线、上划线或删除线），还可以设置它的颜色、样式，例如虚线或波浪线。

这些新的选项大部分都适用于 SVG，但颜色选项由于 SVG 文本具有填充和描边而变得更加复杂，它不是单一的文本颜色。最终，SVG 2 中引入了相应的 `text-decoration-fill` 和 `text-decoration-stroke` 属性。

浏览器已经开始实现对新的文本修饰选项的支持。然而，编写本书时，还没有浏览器实现 SVG 特定的对填充和描边渲染的控制。

例 11-4 中使用的 `text-shadow` 效果是在 CSS2 中初次提出的，但并没有成为最终的规范。现在它已经被包含在 CSS Text Decoration Module Level 3 草案中。编写本书时，它还没有正式作为 SVG 的表现属性而被采用。

第 12 章

有纹理的文本

在第 11 章中，我们已经展示了几个使用图案来填充文本的例子，也在最后强调了一些其他的文本样式。第 2 章中也使用描边的文本来演示了 `paint-order` 属性。

从始至终我们一直都在强调，在 SVG 中填充和描边文本与填充形状类似。

在大多数情况下，确实是这样。但是，对于渲染服务，还是会有一些不同的，因为文本是基于对象边界盒来计算的。本章将更加详细地讲解渲染服务和文本。

如果你正考虑使用形状来替换文本，那熟悉绘制文本和绘制形状之间的差异尤为重要。大多数图形程序提供了一个文本到路径的转换，这样可以保持文本在特定字体中的视觉效果，而不用担心必须把字体和你的图形一起在 Web 上分发，也不用考虑许多浏览器在渲染文本时的 bug（特别是在移动设备上）。依赖于具体的实现方式，这种转换可能因为字母形式的渲染方式不同而产生细微的差别，也可能会有显著的改变。

> 把文本转换为路径会使得文本对于屏幕阅读器，以及想要搜索、复制、翻译文本内容的普通用户不可访问。务必要为任何文本形状提供一个机器可读的替代品。

SVG 规范中有许多控制文本布局的选项，其中有一些是借鉴原来的 CSS2 规范，有一些对于 SVG 来说是新的但 CSS3 中早已采纳，还有一些是 SVG 特有的。有些选项已经在 Web 浏览器中实现了！讨论所有的文本布局选项本身就有一本书[1]，所以本章中的例子除了在指示该属性支持比较差时外，不会讲解太多布局创建的具体细节。

12.1　边界文本

SVG 中绘制在屏幕上的每个元素都有一个边界盒。虽然有些元素并不会绘制，组元素以及其他容器元素比如 `<use>` 和 `<svg>` 同样也有边界盒，这些容器的边界盒用于裁剪、遮罩以及滤镜。边界盒始终是与该元素（包括任何变换）坐标系的轴线对齐的矩形。

对于形状，盒子始终是控制形状基本几何的最小矩形。对于容器，它是包含所有子组件的最小矩形。对于文本，它是包含每个字符的 `em` 盒子的最小矩形。

排版中的 `em` 盒子是每个字符绘制的区域。字体通常占据整个 `em` 盒子的高，以及整个字母间距确定的宽。在许多情况下，实际的字符只占据矩形的一小部分。对于其他字符，字母可能延伸到矩形外，低于或高于文本中同一行的其他字符。最终，你填充的文本区域可能并没有匹配用于缩放渲染服务的对象边界盒。

例 12-1 定义了一个简单的条纹图案，缩放到对象边界盒，且使用它填充四个不同的文本元素。每个元素占据绘制图形的不同空间。图 12-1 显示了它们是如何绘制的。

例 12-1　使用对象边界盒图案填充文本

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="400px" height="300px" viewBox="0 0 400 300"
     xml:lang="en">
    <title>Text Bounding Box</title>
    <style type="text/css">
        text {
            font-size: 50px;
            font-family: sans-serif;
            font-weight: bold;
        }
    </style>
```

注 1：具体来说是 *SVG Text Layout*，作者及出版社与本书完全的一样。

```
    <pattern id="stripes" width="100%" height="20%"
             patternContentUnits="objectBoundingBox">       ❶
        <rect width="1" height="0.1" fill="indigo" />
        <rect width="1" height="0.1" y="0.1"
              fill="royalBlue" />                           ❷
    </pattern>
    <defs>
        <path id="p" d="M250,150 L350,200 250,270" />
    </defs>

    <g fill="url(#stripes)">                                ❸
        <text x="50%" y="1em" dx="-10"
              text-anchor="end">Mixing</text>
        <text x="50%" y="1em" dx="10"
              text-anchor="start">BLOCK</text>               ❹
        <text x="10%" y="3em"
              >Three <tspan x="10%" dy="1.2em"
              >line </tspan><tspan x="10%" dy="1.2em"
              >text</tspan></text>                          ❺
        <text><textPath xlink:href="#p"
              >TextPath</textPath></text>                   ❻
    </g>
</svg>
```

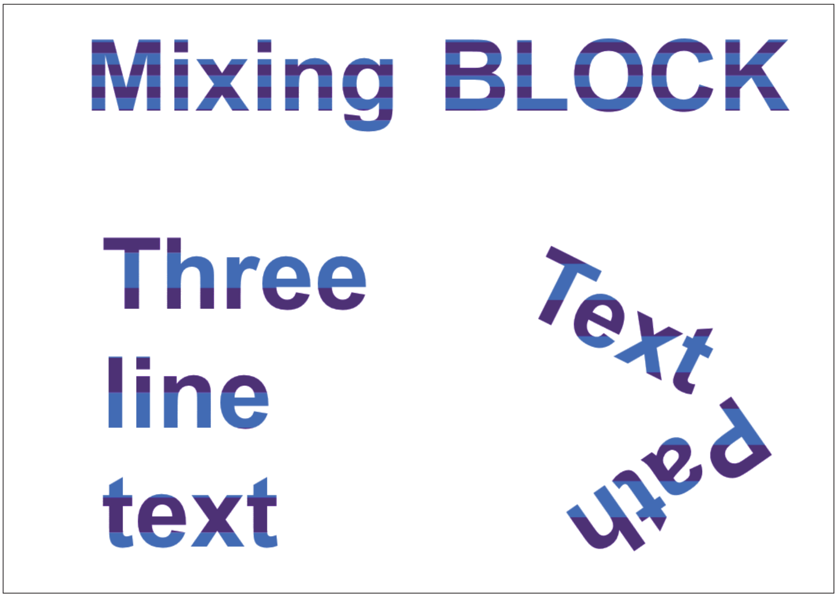


图 12-1：四个使用相同的条纹图案填充的文本元素

❶ 每个图案磁贴延伸到边界盒整个宽度以及高度的五分之一（20%）。

❷ 图案内容也缩放到边界盒。每个条纹是边界盒高度的十分之一（0.1）。

❸ 组元素中应用图案来填充所有的 `<text>` 元素。与往常一样，每个元素将基于自己的边界盒填充。

❹ 前两个元素水平对齐。一个元素字母大小写混合，另一个元素字母全部大写。

❺ 下一个元素使用定位的 `<tspan>` 段把整个文本打破使它跨越三行。

❻ 最后一个文本元素沿着 `<textPath>` 排列，大约占据页面中三行文本的高度。

图 12-1 中要注意的第一件事是，条纹在多行以及 `<textPath>` 文本中的尺寸不同于在单行文本元素中的尺寸。这些元素的边界盒更大，会创建更大的图案磁贴。

如果仔细查看最上面的一行（两个单行元素），你会注意到字母从上到下数并没有五对条纹。你还可以注意到两个单词条纹的尺寸相同，尽管实际上大写字母不会低于基线。这是因为计算边界盒是基于每个字符的整个 em 高，即使使用的字符并没有填满整个空间。

较大的元素同样稍小于它们的边界盒，尽管未使用的部分仅占据较大元素边界盒的一小部分。

> 对于 `<textPath>` 以及其他旋转字符的文本元素，许多浏览器（编写本书时，Blink、WebKit 以及 IE）会随着字母旋转整个图案。对于例 12-1，这将导致有补丁、不连续的条纹。图 12-1 是在 Firefox（v37）中渲染的结果。

相比而言，如果你把文本转换为路径，边界盒将是每个 `<path>` 元素的最小边界盒。图 12-2 显示了使用 Inkscape 中把对象转换为路径的功能的结果。每一个字母都变为一个 `<path>` 元素，因此在形状内从上到下条纹缩放且排列为五对蓝色 - 靛蓝色条纹。

> 在把文本转换为路径之前，例 12-1 中的 SVG 代码首先必须替换用户坐标中所有百分比长度和 em 单位，因为它们目前在 Inkscape（版本 0.91）中都不支持。

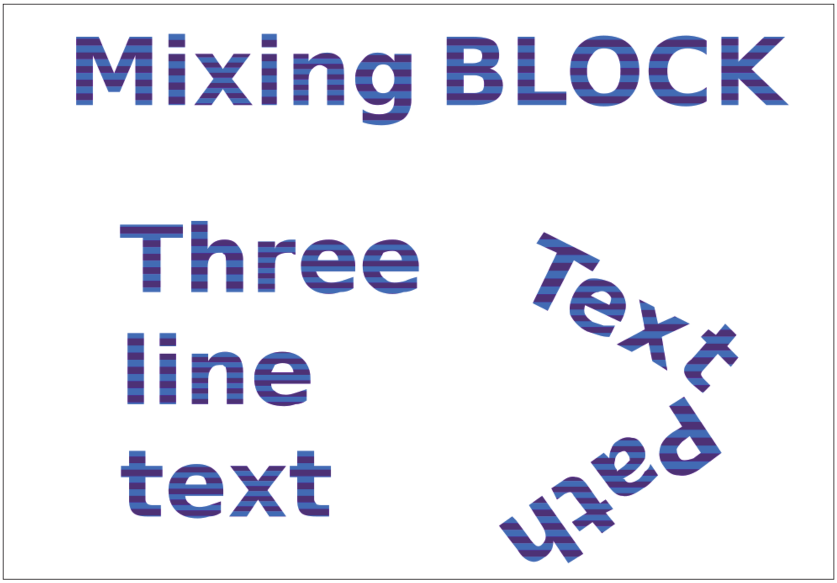


图 12-2：使用相同的条纹图案填充组成单个字母路径

要想让最终的外观与原效果更加接近，可以通过把每个文本块内所有的路径合并为一个拥有多个部分的路径（在 Inkscape 中，可以通过选中创建 `<text>` 元素的组并使用合并路径选项来实现）。但是图案比例依然和原 `<text>` 有一些偏差。文本的边界盒扩展到了字母边界之外，因此它是最终路径的边界盒。

这些可以影响你边界盒图案缩放的因素，将便于你很容易地调整每个图案磁贴的尺寸。不幸的是，例 12-1 中的代码，图案内容也是基于对象边界盒单位度量的，它直接依赖于图案磁贴的尺寸。

即使你不需要图案保持一个固定的宽高比，你也可以使用 `viewBox` 给内容建立一个独立于磁贴尺寸的完整的坐标系。`preserveAspectRatio` 设置为 `none` 会使正方形被拉伸去填满整个磁贴，而忽略磁贴尺寸的所有变化。就像对象边界盒单位那样，缺乏宽高比控制将创建一个扭曲的坐标系，但是对于长方形条纹是没有问题的。



正如第 11 章中提到的那样，40 版本之前的 Firefox 不能正确渲染图案上的 `viewBox`。

例 12-2 展示了对于条纹图案它是如何工作的。为了演示图案内容和图案磁贴之间的独立性，通过使用 xlink:href 复制图案内容创建了两种不同的图案——一种粗条纹，一种细条纹。图 12-3 显示了两个版本的图案。

例 12-2　不保持宽高比，使用 viewBox 创建可缩放的边界盒图案

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="400px" height="70px" viewBox="0 0 400 70"
     xml:lang="en">
    <title>Adjustable Stripes</title>
    <style type="text/css">
        text {
            font-size: 50px;
            font-family: sans-serif;
            font-weight: bold;
        }
    </style>
    <pattern id="thick-stripes" width="100%" height="40%"
             viewBox="0 0 2 2" preserveAspectRatio="none">
        <rect width="2" height="1" fill="indigo" />
        <rect width="2" height="1" y="1"
              fill="royalBlue" />
    </pattern>
    <pattern id="thin-stripes" height="12.5%"
             xlink:href="#thick-stripes" />
    <text x="50%" y="1em" dx="-10" fill="url(#thick-stripes)"
          text-anchor="end">Stripy</text>
    <text x="50%" y="1em" dx="10" fill="url(#thin-stripes)"
          text-anchor="start">Stripy</text>
</svg>
```



图 12-3：使用不同尺寸条纹图案填充的文本元素

图案中的每个长方形占据 viewBox 高度的一半以及整个宽度。边界盒高度范围之内水平条纹的数量因此只由图案上的 height 属性控制，且很容易在引用第一个图案的第二个 <pattern> 元素上进行覆盖。

12.2　中途切换样式

虽然例 12-1 中的每个 `<text>` 元素都会创建自己绘制时的对象边界盒，但多行文本中的单个 `<tspan>` 元素不会。也就是说单个 `<tspan>` 元素可以设置与剩余文本不同的填充值。

> 如果 `<tspan>` 的填充使用的是对象边界盒缩放，它是基于整个 `<text>` 元素来缩放的，而不是它本身。

例 12-3 演示了这种使用边界盒渐变来填充 `<text>` 元素中三个单独的 `<tspan>` 元素的效果。它还提供了一个可读性稍强的、使用图案来填充文本的例子。它没有使用粗条纹来填充，而是使用一个更加精细的纹理来重新创建粉笔写在黑板上的效果。图 12-4 显示了最终效果。

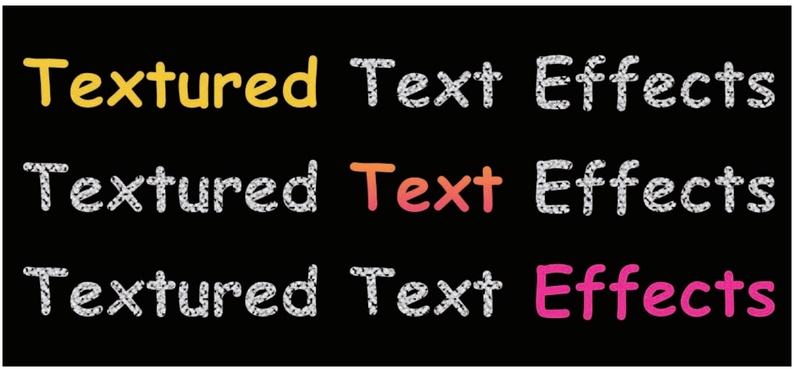


图 12-4：图案和渐变填充的多行文本

例 12-3　使用对象边界盒图案和渐变来填充文本

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="10cm" height="5cm"
     xml:lang="en">
    <title>Chalkboard Text</title>
    <style>
        text {
            font-family: cursive;
            font-weight: bold;
            font-size: 9mm;
```

```
            font-size-adjust: 0.54;                    ❶
        }
        .chalk {
            fill: url(#chalk-texture);
        }
        .chalk-marks {
            fill: white;
            fill-opacity: 0.8;
            stroke: silver;
            stroke-width: 0.2px;
        }
        .highlight {
            fill: url(#highlight-gradient);
        }
        .blackboard {
            fill: black;
        }
    </style>
    <pattern id="chalk1" class="chalk-marks"
             patternUnits="userSpaceOnUse"
             width="9" height="6">                              ❷
        <circle r="1" cx="0" cy="3" />
        <circle r="1" cx="1" cy="5" />
        <circle r="1" cx="2" cy="1" />
        <circle r="1" cx="3" cy="2" />
        <circle r="1" cx="4" cy="5" />
        <circle r="1" cx="5" cy="4" />
        <circle r="1" cx="6" cy="0" />
        <circle r="1" cx="7" cy="6" />
        <circle r="1" cx="8" cy="3" />
    </pattern>
    <pattern id="chalk2" xlink:href="#chalk1"
             x="2" width="5" height="5"/>                      ❸
    <pattern id="chalk3" xlink:href="#chalk1"
             y="4" width="7" height="7"/>
    <pattern id="chalk-texture" patternUnits="userSpaceOnUse"
             width="315" height="210">                          ❹
        <rect fill="url(#chalk1)" width="315" height="210"/>
        <rect fill="url(#chalk2)" width="315" height="210"/>
        <rect fill="url(#chalk3)" width="315" height="210"/>
    </pattern>

    <linearGradient id="highlight-gradient" x2="0" y2="1">
        <stop stop-color="gold" offset="0.3" />                 ❺
        <stop stop-color="deeppink" offset="0.7" />
    </linearGradient>
    <rect class="blackboard" width="100%" height="100%" />
    <text y="5mm" class="chalk" textLength="29cm">              ❻
```

```
        <tspan x="3mm" dy="1em"><tspan
               class="highlight">Textured</tspan>          ❼
            Text Effects</tspan>
        <tspan x="3mm" dy="1.6em">Textured
            <tspan class="highlight">Text</tspan> Effects</tspan>
        <tspan x="3mm" dy="1.6em">Textured
            Text <tspan class="highlight">Effects</tspan></tspan>
    </text>
</svg>
```

❶ 你不必使用 px 或 pt 来设置文本的大小：SVG 所会缩放到使用合适的度量单位，文本也是如此。`font-size-adjust` 的设置是为了保证无论实际使用什么字体，字体的大小可以保持大致相同。0.54 的值是基于 Comic Sans MS 的，它是 Windows 计算机上大多数浏览器默认的 `cursive` 字体。

❷ 基础的粉笔图案是通过一小块图案磁贴内九个位置不规律的白色和灰色点来创建的。圆的样式是在 `<pattern>` 上的 `chalk-marks` 类中设置的。

❸ 为了使图案更加紧密、更加不规则，它被复制了两次，但每个图案磁贴的尺寸和偏移都不一样。

❹ 聚合的图案是通过把三个粉笔标记图案放在一起布局来创建的。聚合的图案使用 `userSpaceOnUse` 单位来定义一个更大的图案磁贴，这使它足够容纳重复偶数次的各层图案。

❺ 用于高亮 `tspan` 的渐变是一个简单的缩放到对象边界盒（`gradientUnits` 默认值）的垂直线性渐变。约前三分之一是纯金色（0.3 的偏移），后三分之一（偏移大于 0.7）是纯粉色，且在两者之间有一个过渡。

❻ 文本在一个 `<text>` 元素内。`textLength` 属性用于调节三行文本的总长度使其略小于三倍 SVG 的宽度。

❼ 三个 `<tspan>` 元素使文本折为三行。每一行中，有 `highlight` 类的 `<tspan>` 使包含在其中的单词使用不同的填充。

图 12-4 内渐变填充的单词非常明显地展示了整个 `<text>` 元素作为边界盒时是什么样的。而且，在粉笔纹理中这同样很明显。如果每一行的文本分别在一个元素内，使用相同的图案填充时，填充中的裂缝和不规则的点将在完全相同的位置。本例子中由于每个组成它的图案都是基于其自己的大小来重复的，且恰好没有完全匹配一行文本的高度，所以结果看起来是随机的，如图 12-5 所示，图中放大显示了前两行中的最后一个单词。



图 12-5：伪随机纹理图案的特写视角

例 12-3 使用浏览器默认的 `cursive` 字体。通常情况下，这种做法并不推荐，因为这些字体在不同浏览器和操作系统之间可能会发生很大变化。通常做法是给定一个 `font-family` 名字列表，或者使用 Web 字体来确保正确的字体是可用的。然而，有时这些方法都不起作用，所以倡导渲染一致性的其他代码属性也值得我们去关注，比如 `textLength` 属性和 `font-size-adjust` 样式。

> `textLength` 属性（把文本压缩或拉伸到给定的长度）在浏览器中有很多 bug，且在所有旧版本的 SVG 工具中都不支持。这里使用它的方式（在父级 `<text>` 元素上定义）在 Firefox 和 IE 中支持。换句话说，Blink 和 WebKit 浏览器不会调整 `<tspan>` 中的内容，除非 `textLength` 属性设置在每个子元素上。Firefox 会忽略 `<tspan>` 上的 `textLength`，且 IE 在长度调整时不能正确居中或两端对齐文本。
>
> `font-size-adjust` 属性，用于使浏览器保持字体的 `ex-height` 而不是 `em-height`（通过指定两者之间的比率），目前只有 Firefox 浏览器支持。

在符合标准的浏览器中，这些设置可以确保即使使用的字体完全不同，也可以保持整体的布局，如图 12-6 所示。在其他浏览器中，文本可能会没有填充整个宽度，或距离边缘太近。



图 12-6：不同字体中，图案和渐变填充的多行文本

CSS 与 SVG

使用图形填充文本

例 11-4 中图片填充的文本和例 12-3 中的有纹理的文本对于标题或其他大型文本有很好的效果。为了在非 SVG 文本中实现该效果，基于 WebKit 的浏览器（包括 Safari、Chrome 以及新版本的 Opera）都支持不标准的属性 `-webkit-background-clip`，因此可以用它来创建类似的效果。

标准的 `background-clip` 属性允许把背景裁剪到元素的 `content-box`、`padding-box` 或 `border-box`。试验阶段的 WebKit 选项还支持把背景裁剪到元素的文本内容。结合透明文本（使用 WebKit 特定的 `webkit-text-fill-color` 设置，所以对其他浏览器没有影响）可以创建图片填充的文本。

编写本书时，还没有现成的或处于草案阶段的 Web 标准来给 SVG 之外的文本填充图片或图案。然而，过去一直在反复讨论对 CSS 样式的文本使用 `fill` 和 `stroke`。这也可能通过元素及背景图片的混合和合成新选项来实现相似的效果。

第 13 章

绘制线条

之前渐变和图案的例子几乎都是使用 `fill` 属性来把它绘制到形状上的。其实我们本书中提到的渲染服务同样可以用于 `stroke` 属性。

在描边上使用渲染服务会造成一些新的难题，这就是我们把绘制线条单独作为一章的原因。这里主要介绍难点，也会给出一些通过渲染描边来创建的独特效果的例子。我们还会介绍在 SVG 2 中提出的一些新特性，这些特性让线条绘制更简单。

13.1 超出边缘的部分

我们之前简要的提过，描边不一定只能使用纯色。和 `fill` 属性一样，`stroke` 属性可以使用一个 `url()` 函数来通过渲染服务（渐变或图案）的 `id` 值来引用它。你也可以使用一个备用的颜色，以防渲染服务出现问题，而且很快你就能看到这种备用颜色的用武之地。

在要使用渲染服务时，你可能会想起描边是一个次要形状，依赖于定义它的元素，而不是用钢笔或刷子画的线。目前图案和渐变还不能沿着路径绘制。与绘制形状的填充类似，描边是从类似墙纸的渲染服务内容中裁剪出来的。

在描边中使用渲染服务时，有两个地方让设计者感到头疼：

- 渲染服务使用的 `objectBoundingBox` 单位不包括描边区域
- 所有的渲染服务都会创建一个不受形状或描边方向影响的矩形绘制区域

这些难题并非总是令人费解的。对于许多形状，渐变或图案描边都能工作得很好。例 13-1 使用第 6 章中基本的水平红蓝渐变来绘制一个描边很宽的矩形，如图 13-1 所示。

例 13-1　在矩形中使用渐变来绘制描边

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4in" height="1.5in" >
    <title xml:lang="en">Gradient on a Stroke</title>
    <linearGradient id="red-blue" >
        <stop stop-color="red" offset="0"/>
        <stop stop-color="lightSkyBlue" offset="1"/>
    </linearGradient>
    <rect width="80%" height="50%" x="10%" y="25%"
          stroke-width="20%" stroke="url(#red-blue)"
          fill="none" />
</svg>
```



图 13-1：使用线性渐变描边的矩形

渐变矢量的长短是根据长方形的几何尺寸确定的，而不是描边的尺寸，但是这种渐变平滑的比较难以判断。如果将 spreadMethod 的值更改为 repeat，这就会变得很明显，如图 13-2 所示。每条边上描边的一半是在渐变矢量外部，而在重复的区域内。



图 13-2：使用重复线性渐变描边的长方形

聚焦未来
多个描边

我们已经介绍了 SVG 中如何层叠多个填充，如何创建多个堆叠的渐变或图案来突出单独设置的纯色。描边中也将实现同样的功能甚至会有更多功能。因为每个描边可能会有不同的宽度或线条图案，甚至纯色描边也可以有效地分层。

确切的细节在编写本书时还没有最终敲定，但是 SVG 2 很可能会采用类似于 CSS 分层背景的语法。不管任何属性都将指定为一个值列表，如果一个属性的列表长度和主（stroke）属性不同，它将会根据需要进行重复。所以，如果想要所有层都有 0.5 的不透明度，你只需要指定该值一次。

与填充类似，描边的另一种新内容是将支持直接在 stroke 属性内使用 CSS 图片数据类型——渐变函数以及引用其他图片文件。和 SVG 渐变一样，最终描边区域是从渐变填充的矩形中裁剪，而不是从沿着描边方向的渐变上裁剪。

图 13-2 中锐利的重复边缘并不是偶然的。因此你可能会认为 objectBoundingBox 单位并不是一个很大的障碍。确实如此，除非你是给线条而非形状描边。

13.2 空盒子

例 13-2 使用相同的红蓝线性渐变给一系列不同方向的直线描边。因为直线不能被填充，通常使用一条粗的渐变描边来实现我们想要的效果。

例 13-2　在直线上使用渐变来绘制描边

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4in" height="3in" >
    <title xml:lang="en">Gradient on Straight Lines</title>
    <linearGradient id="red-blue" >
        <stop stop-color="red" offset="0"/>
        <stop stop-color="lightSkyBlue" offset="1"/>
    </linearGradient>
    <g fill="none" stroke-width="0.5in"
       stroke="url(#red-blue) purple" >              ❶
        <line x1="10%" x2="90%" y1="10%" y2="10%" /> ❷
        <line x1="90%" x2="90%" y1="25%" y2="75%" />
        <line x1="90%" x2="10%" y1="90%" y2="90%" />
```

```
            <line x1="10%" x2="10%" y1="75%" y2="25%" />
            <line x1="30%" x2="70%" y1="25%" y2="75%" /> ❸
            <line x1="70%" x2="30%" y1="25%" y2="75%" />
        </g>
    </svg>
```

❶ 样式属性通过表现属性应用在包含每条线的组元素上。描边值包括一个对渐变和备用的纯色的引用。

❷ 最初的四条线括起绘制区域：横穿过顶部，然后沿着右边向下，然后沿着底部水平返回，最后沿着左边向上。

❸ 最后两条是对角线，一条从左上到右下，另一条从右上到左下。

代码看似简单，但如图 13-3 所示，结果并不是我们期望的。

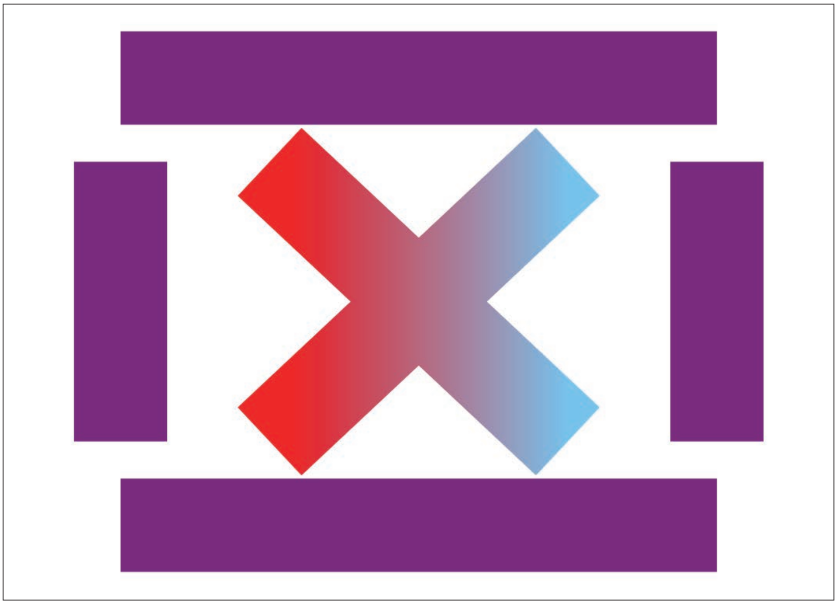


图 13-3：使用线性渐变或备用的颜色描边的线

为什么这么多线是纯紫色描边呢？这是因为完全水平或垂直的线没有边界盒区域。没有边界盒区域就意味着没有边界盒渐变。如果我们没有指定备用的颜色，描边将一直都不可见。

直觉上这似乎是渐变的极端情况。毕竟，即使水平线没有任何高度，但是它有宽度。那不正好是水平线性渐变吗？对于垂直线，不应该至少一边使

用红色填补，另一边使用蓝色填补吗？

难点的来源是渲染服务的实现方式，因为渲染的矩形块被变换到适应边界盒。变换导致任何维度都坍塌为 0，这造成浏览器内部数学计算时产生除数为零的错误。为了避免这个问题，SVG 规范中把零宽度或零高度的边界盒当作边界盒渲染服务的一种错误。浏览器会使用相同的方式响应错误，如果找不到渲染服务，就使用备用的颜色。

相比之下，对角线是使用渐变绘制的，即使线条依然没有填充区域，因为边界盒是与坐标系轴线对齐、可以包含对象的最小矩形。对于有角度的线，它就是矩形的对角线。虽然两条对角线绘制的方向相反，但它们有相同的边界盒，所以两者之间的渐变是一致的。

当人们思考线条上的渐变时，他们通常想到的是像图 13-4 中显示的那样。每个渐变从线的一端到另一端，方向和线条绘制的方向一致。

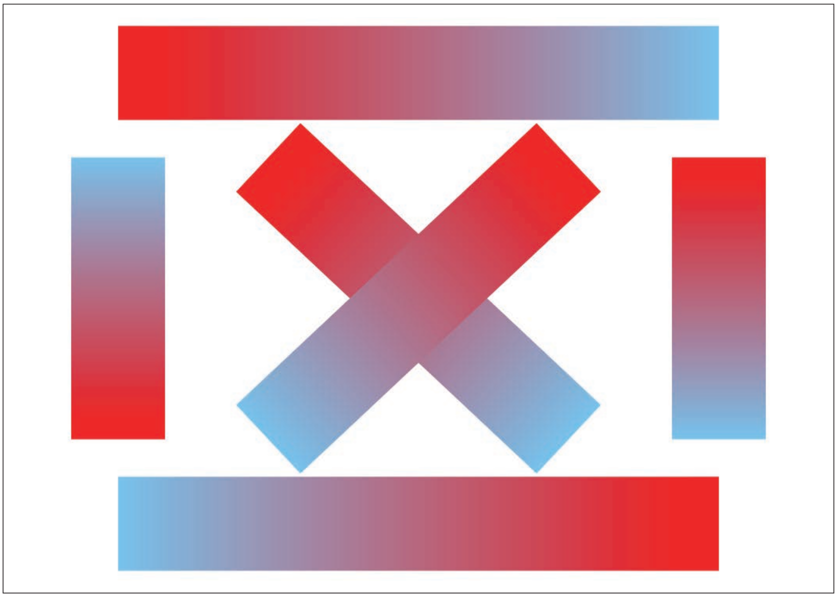


图 13-4：使用方向匹配的线性渐变来描边的线条

你可能已经猜到了，SVG 可以创建图 13-4 的效果。然而，它需要比例 13-2 中更多的标记。你不能给渐变使用对象边界盒单位，所以需要使用 `userSpaceOnUse` 渐变。然后需要让渐变矢量与线条的位置、方向以及长短都一一匹配。

有两种可以实现的方式。例 13-3 中显示了图 13-4 使用的代码。它给每个 <line> 都创建了一个单独的 <linearGradient>，并设置渐变矢量的位置属性与对应的线相同。

例 13-3　匹配线性渐变和直线

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="3in" >
    <title xml:lang="en">Gradients Made to Measure
                        for Straight Lines</title>
    <linearGradient id="red-blue"
                    gradientUnits="userSpaceOnUse">       ❶
        <stop stop-color="red" offset="0"/>
        <stop stop-color="lightSkyBlue" offset="1"/>
    </linearGradient>
    <g fill="none" stroke-width="0.5in" >                 ❷
        <linearGradient xlink:href="#red-blue" id="g1"
              x1="10%" x2="90%" y1="10%" y2="10%" />      ❸
        <line x1="10%" x2="90%" y1="10%" y2="10%"
              stroke="url(#g1) purple"/>                  ❹
        <linearGradient xlink:href="#red-blue" id="g2"
              x1="90%" x2="90%" y1="25%" y2="75%" />
        <line x1="90%" x2="90%" y1="25%" y2="75%"
              stroke="url(#g2) purple"/>
        <linearGradient xlink:href="#red-blue" id="g3"
              x1="90%" x2="10%" y1="90%" y2="90%" />
        <line x1="90%" x2="10%" y1="90%" y2="90%"
              stroke="url(#g3) purple"/>
        <linearGradient xlink:href="#red-blue" id="g4"
              x1="10%" x2="10%" y1="75%" y2="25%" />
        <line x1="10%" x2="10%" y1="75%" y2="25%"
              stroke="url(#g4) purple"/>
        <linearGradient xlink:href="#red-blue" id="g5"
              x1="30%" x2="70%" y1="25%" y2="75%" />
        <line x1="30%" x2="70%" y1="25%" y2="75%"
              stroke="url(#g5) purple"/>
        <linearGradient xlink:href="#red-blue" id="g6"
              x1="70%" x2="30%" y1="25%" y2="75%" />
        <line x1="70%" x2="30%" y1="25%" y2="75%"
              stroke="url(#g6) purple"/>
    </g>
</svg>
```

❶ 基础的渐变包含用于模板的颜色结点。gradientUnits 的值设为 userSpaceOnUse，它将会被引用它的渐变继承。

❷ 不必在组元素上设置 stroke 的值，每个元素都需要一个单独的描边属性来引用定制的渐变。

❸ 每个 <linearGradient> 元素都有一个引用渐变模板的 xlink:href 属性、一个唯一的 id 以及与对应的 <line> 相匹配的 x1、x2、y1、y2 等属性。

❹ 每条线都有一个通过表现属性设置的正确的描边渐变。备用的颜色是为了以防万一，但不会被使用，除非我们产生了错误。

例 13-3 中代码的主要不足之处是有太多重复。例如，<line> 和 <linear-Gradient> 上重复的属性在你想要改变它的位置时都要更新两次。但如果你第一时间就使用基于数据的脚本来生成线条，这种结构是相当有效的。你可以创建一个把数据设置为元素上各种属性的函数，然后运行两次，一次用在线上，一次用在渐变上。

> 如果你的 <line> 元素部分位置属性使用的是默认值，一定要记住 <linearGradient> 上 x2 的默认值与之不同（100%，而不是 0）。

例 13-3 中所用方法的另一个限制是 DOM 元素的数量明显增加了。如果动态图形中有大量的线条和渐变，网页会变得很慢。

调整渐变和线的另一种方式是使用单个渐变，然后让所有的线都沿着它绘制。然后，使用变换来把它们调整到合适的尺寸和位置。例 13-4 中我们使用了这种方式。由于 SVG 1.1 中变换必须使用用户单位定义，不能使用百分比，绘制区域必须重新使用 viewBox 定义并调整数值。结果，如图 13-5 所示的最终 SVG 并没有和图 13-4 完全匹配，但是非常接近。

例 13-4　变换线条的绘制来匹配线性渐变

```
<svg xmlns="http://www.w3.org/2000/svg" xml:lang="en"
     width="4in" height="3in" viewBox="0 0 100 75">                    ❶
    <title xml:lang="en">Userspace Gradient on
                         Transformed Straight Lines</title>
    <linearGradient id="red-blue" gradientUnits="userSpaceOnUse">   ❷
        <stop stop-color="red" offset="0"/>
        <stop stop-color="lightSkyBlue" offset="1"/>
   </linearGradient>
    <g fill="none" stroke-width="12"
       stroke="url(#red-blue) purple" >                                ❸
        <line x2="100%"
              transform="translate(10,7.5) scale(0.8,1)" />           ❹
        <line x2="100%"
              transform="translate(90,18.75)
                         rotate(90) scale(0.375,1)" />               ❺
        <line x2="100%"
              transform="translate(90,67.5) scale(-0.8,1)" />         ❻
```

```
        <line x2="100%"
              transform="translate(10,56.25)
                         rotate(-90) scale(0.375,1)" />
        <line x2="100%"
              transform="translate(30,18.75)
                         rotate(45) scale(0.5315,1)" />
        <line x2="100%"
              transform="translate(70,18.75)
                         rotate(135) scale(0.5315,1)" />
    </g>
</svg>
```

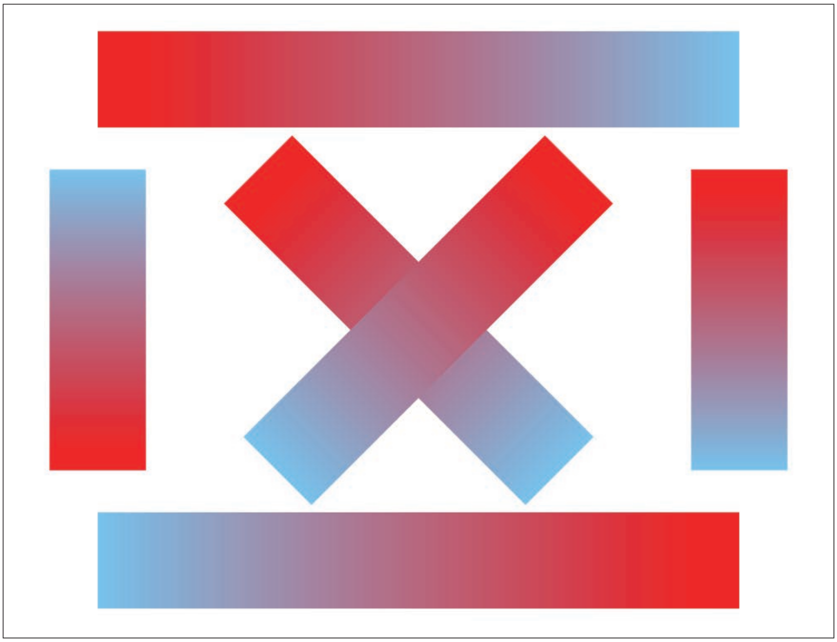


图 13-5：使用渐变给线描边，并把它变换到合适的位置

❶ viewBox 使用 100 个单位的宽度，所以水平上的百分比可以直接转换为用户单位值。但垂直百分比需要缩放到 70 个单位高度。

❷ 我们只需要一个使用 userSpaceOnUse 单位的渐变。渐变矢量使用默认的排列方向：水平从 0 到 100%。

❸ 单独的一个 stroke 值设置在包含所有线的组元素上。同样，备用颜色是以备不时之需。用户空间渐变应该始终提供有效的结果。

❹ 每个 `<line>` 元素都有相同的位置属性：`x2="100%"` 用于匹配 `<linear-Gradient>` 上的默认属性。线条的实际尺寸和位置是通过 `<transform>` 属性控制的。

❺ 变换按照一个特定的顺序排列，以确保它们不会以不可预见的方式互相影响：首先，原点（线的起点）被移动到一个合适的位置，然后把它旋转到正确的方向，最后缩放它的长度（*x* 轴是在变换后的坐标系中）。

❻ 为了替换 180 度的旋转，右到左的渐变使用一个负的缩放值来创建。

通过这种方式，你不需要创建任何额外的元素。但你必须完全重新定义图形的几何形状。如果你事先知道渐变是图形必不可少的部分，并且可以提前规划好，这可能并不是一个很大的障碍。但像这样重写一个图形——使用变换来替换通过位置属性定义的代码——通常带来的麻烦多于好处。

聚焦未来

创建一个描边边界盒

正如你现在知道的，给描边使用渲染服务的一个最大的限制是对象边界盒不包含描边。SVG 2 中一个最值得期待的变化是可以指定包含描边区域的不同边界盒。

边界盒的使用是在使用渲染服务时指定的，而不是在图案或渐变元素上定义。这将允许相同的渐变或图案可以恰好适应填充边界盒、描边边界盒、甚至用户空间。但这些依然是“盒子”区域：与坐标系水平和垂直的轴对齐的矩形。要创建沿着线的方向的渐变，你依然需要例 13-3 或例 13-4 中使用的方法。实际上沿着线条曲率的渐变需要 SVG 当前的渲染工作进行更深层的改变，且短期内还不可预期。

定义渲染服务所使用的盒子的确切语法在编写本书时还没有最终敲定。但预期是直接在 `stroke`（或 `fill`）的简写属性中定义，或作为一个单独的样式属性。

13.3 使用坐标空间

除了作为沿着直线的渐变的替代品（虽然并不完美），用户空间描边渐变有着更多的应用。在数据可视化中，它们可以创建根据值来改变数据线外观的效果。例 13-5 演示了这种效果。它创建了一个通用的状态监控仪表盘，

其中有一条线用于标记变化值不应达到 100%：随着线条距离最大值越近，它的颜色从绿色变为黄色再变到红色。

图形的基本框架——包括渐变——定义在标记内，而数据是通过脚本插入的。在实践中，数据可能来自某种 Web 服务器。在该例子中，数据是随机生成的。图 13-6 显示了一种可能的数据图案。

例 13-5　使用渐变来给折线图添加信息

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4in" height="3in"
     xml:lang="en">
    <title>Stroked Gradient as Status Indicator</title>
    <svg x="0.6in" y="0.5in" width="3.6in" height="2in"
         viewBox="0 0 180 100"
         style="overflow: visible; font: 10px sans-serif;">    ❶
        <linearGradient id="status" y1="100%" y2="0%" x2="0%"
                        gradientUnits="userSpaceOnUse">            ❷
            <stop stop-color="limegreen" offset="0.4"/>
            <stop stop-color="yellow"    offset="0.8"/>            ❸
            <stop stop-color="red"       offset="0.95"/>
        </linearGradient>

        <rect stroke="black" fill="dimGray"
              width="100%" height="100%" />
        <g text-anchor="end" dominant-baseline="middle"
           transform="translate(-2,0)">                            ❹
            <desc>Y-axis tick labels</desc>
            <text y="100">0%</text>
            <text y="80">20%</text>
            <text y="60">40%</text>
            <text y="40">60%</text>
            <text y="20">80%</text>
            <text y="0">100%</text>
        </g>
        <g text-anchor="middle">
            <text x="50%" dy="-1em" font-size="12px"
                  text-decoration="underline"
                  >Status monitor</text>
            <text x="50%" y="100%" dy="1em">Time</text>
        </g>

        <polyline id="dataline" stroke="url(#status)"
                  stroke-linejoin="bevel"
                  stroke-width="2.5" fill="none" />              ❺
    </svg>
    <script><![CDATA[
(function(){
```

```
        var n=19,
            dx=10,
            maxY = 100;                                           ❻

        var data = new Array(n),
            points = new Array(n);
        for (var i=0; i<n; i++) {
            data[i] = [i, Math.random()];
            points[i] = [i*dx, maxY * (1 - data[i][1])];          ❼
        }

        var dataline = document.getElementById("dataline");
        dataline.setAttribute("points", points.toString() );      ❽
    })()
    ]]> </script>
    </svg>
```



图 13-6：使用描边渐变来强调变化值的数据图表

❶ 嵌套的 <svg> 用于创建一个完全匹配图表中数据区域大小的用户空间坐标系。viewBox 应用了一个自定义坐标系，包括 100 个单位的高，这样可以很容易地转换为百分比值。

❷ <linearGradient> 绘制在用户空间坐标系中，它的渐变矢量是从下到上的。

❸ 渐变的前 40% 是纯绿色，然后逐渐变为黄色，再到红色，95% 之后是纯红色。

❹ 使用纯色的背景填充数据区域，文本标记偏移到数据区域的外部（overflow 设置为 visible）。通过把标签包含在嵌套的 SVG 内，这样可以基于 viewBox 坐标系来设定它们的位置，甚至可以打印在宽度和高度区域之外。

❺ 数据线是一个 <polyline> 元素。它的表现样式（包括渐变的描边）在标记中指定，但它没有 points 属性，所以直到脚本运行时它才会绘制。id 属性使它便于在脚本中访问。

❻ 脚本内，初始化的一组变量用于控制数据和坐标系之间的比例：n 是图形中合适的数据值的个数，dx 是它们之间的水平间隔（用户单位），maxY 是垂直范围（0%~100%）应该缩放的数值。

❼ data 数组包含原始的数据（在这里，包括一个索引值以及一个 0 到 1 之间的随机数）。points 数组包含这些数据点缩放后的坐标，使用控制变量来缩放数据以及转化 *y* 坐标，所以最终 0% 在底部而 100% 在顶部。

❽ points 二维数组使用默认的 toString() 方法转换为一个逗号分隔的字符串，并设置为折线的 points 属性。

有效地使用 userSpaceOnUse 渐变来传递数据的关键是精确地匹配用户空间坐标系与数据区域。嵌套的坐标系允许 viewBox 完全匹配数据区域，而且还为区域外的标签留有空间。

> 在 IE 以及 WebKit/Blink 浏览器中，要想把渐变正确地缩放到数据区域，<linearGradient> 必须是嵌套的 SVG 的子元素。正如已经提到的，这些浏览器使用渲染服务的父级坐标系作为 userSpaceOnUse 而不是与之关联的绘制的形状的坐标系。

用户空间渲染的描边的视觉效果类似于一个连续的矩形渐变遮罩在线条上，仅仅显示线条的区域。然而，就像我们在第 11 章中讨论的图像填充的文本那样，渐变裁剪为线条和线条使用渐变渲染之间在结构上有很大差异。如果你想在数据图表中添加交互效果，你往往希望线条本身是交互的对象，而不是大部分都不可见的矩形区域。

这种效果不一定非要限制在实际的数据图形内，你可以使用一个纹理图案或图片作为穿透描边区域露出的内容。

13.4　有图案的线条

图案描边和渐变描边工作方式基本一致。但由于图案在垂直和水平上严密地重复磁贴，当描边方向和渲染服务没有对齐时会更加明显。

与渐变一致，建议使用 `userSpaceOnUse` 的方法来避免 `objectBoundingBox` 单位带来的一些问题。如果确认形状有一个有效的边界盒（即它不是垂直或水平的直线），你仍然可以使用边界盒图案，但图案的缩放是基于填充区域的而非描边区域的缩放。

例 13-6 使用了第 10 章和第 11 章中各种各样的图案来给一些基本的形状描边——配有不同程度的效果，如图 13-7 所示。

例 13-6　使用渐变来给折线图添加信息

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="4.3in" height="4.3in" viewBox="0 0 400 400">
    <title>Patterned Strokes</title>
    <defs>
        <!-- pattern definitions clipped -->                ❶
    </defs>
    <g style="stroke-width: 40px; fill: lightSkyBlue;" >    ❷
        <rect x="20" y="20" width="360" height="360"
              stroke="url(#pinstripe) gray" />              ❸
        <polygon points="200,30 370,200 200,370 30,200"
              stroke="url(#scales-pattern) green"/>         ❹
        <circle cx="200" cy="200" r="90"
              stroke="url(#gradient-pattern) peachPuff"/>   ❺
    </g>
</svg>
```

❶ 图案（以及组成它的渐变）都是直接从例 10-5、例 10-6 和例 11-1 中复制过来的，所以这里没有重复打印。

❷ 所有形状在淡蓝色的填充之上都有一个粗的、40px 宽的描边。

❸ 长方形使用例 10-6 中的 pinstripe 图案。

❹ 菱形的 `<polygon>` 使用的是例 10-5 中的鱼鳞图案。

❺ 圆形使用例 11-1 中的分层布局。与其他形状一样，它被赋予了一个备用的颜色，以备不时之需。

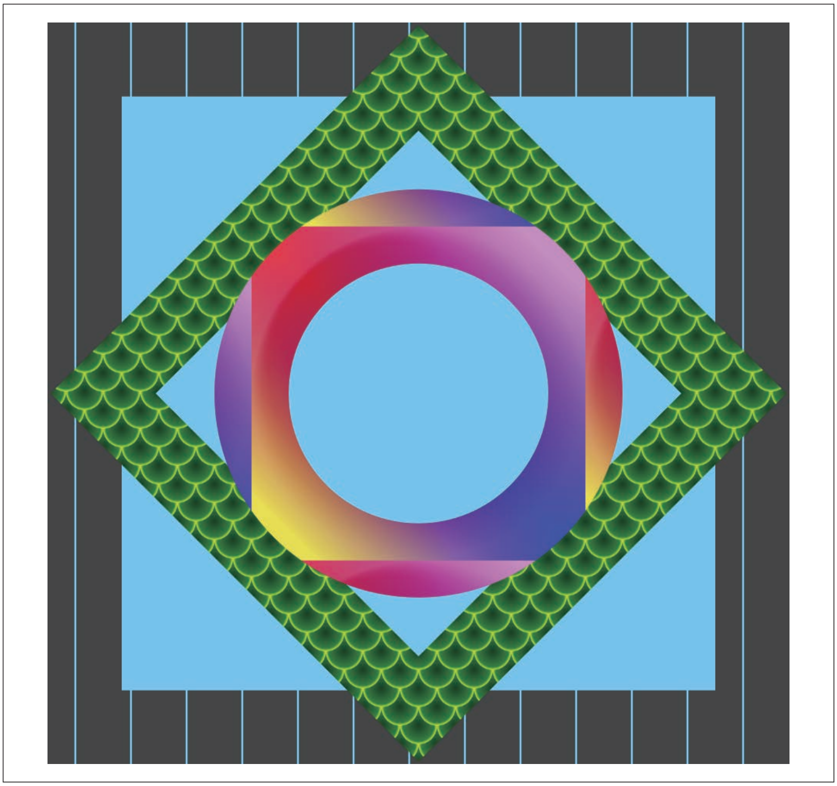


图 13-7：使用各种各样图案进行描边的形状

小的、重复的鱼鳞图案在宽的描边上看起来印象相当深刻。pinstripes 不一定有你想要的效果：竖条纹会跨过正方形水平方向的描边，但在垂直方向上沿着描边竖直向下。分层的描边创建了一个相当不寻常的效果，渐变都是使用边界盒图案来编写的，因为边界盒不会填充整个描边区域，它们在每条边上平铺，创建了颜色锐利变换的效果。

与渐变一样，我们也有解决这些问题的办法。例如，一些条纹的描边效果可以使用虚线描边来创建。为了避免在使用对象边界盒图案描边时的平铺，你可以设置 `x`、`y`、`width` 和 `height` 属性来定义一个足以包括描边区域的图案磁贴（即 `x` 和 `y` 是负的值，`width` 和 `height` 的值都大于 100%）。所有这一切都使多重描边和边界盒描边在将来变得更加容易。

第 14 章

动画

SVG 颜色、渐变和图案可以通称为绘画，但是它与油画和水彩画有一个非常重要的不同：SVG 绘画可以移动。

Web 浏览器中的 SVG 是动态的。我们可以给它添加无限循环或响应用户交互的动画。

动画使 SVG 绘制属性和渲染服务在某些方面变得更加复杂。本章总结了一些主要的问题以及与之对应的解决办法。起初我们回顾了 SVG 添加动画的不同方式，并以动态填充颜色作为示例。然后主要讲了使用渲染服务添加动画的两种情况：给许多绘制元素添加同步的动画，或给单个元素添加动画而不影响其他元素。

截图来自动画示例，但更直观的体验需要在 Web 浏览器中运行代码。

对 SVG 动画完整的讨论可能需要几本书才能讲完，所以本章没有试图描述所有的可选项和语法。相反，我们着重于颜色和渲染服务所特有的方面。如果你还不熟悉 CSS、SMIL 以及 JavaScript 动画，你可能需要查阅其他资料才能完全理解代码是如何创建最终效果的。

14.1　动画选项

给 SVG 图形添加动画的方法有三种：

- 在标记内包含动画元素（`<animate>`、`<set>`、`<animateTransform>` 以及 `<animateMotion>`）来修改其他元素；
- 在图形的样式中添加 CSS 动画或过渡属性；
- 使用 JavaScript 依次操作图形的样式或属性。

这三种方法至少都提供了创建交互动画的可能。动画元素可以由用户的 click 或 mouseover 等事件来触发或修改起始和终止状态。CSS 动画和过渡可以通过交互的伪类（`:hover`，`:focus`，`:active`）来触发。JavaScript 当然可以用于各种用户交互和程序。

使用哪种方式来给 SVG 添加动画很大程度上取决于你准备如何使用 SVG。当 SVG 作为图片嵌入在其他网页中时，无论是通过 `<img>` 标签、CSS 背景图像还是其他相似的属性来引入，JavaScript 都不会运行。此时，SVG 代码必须包含在网页中或通过有交互的 `<object>` 来嵌入。JavaScript 也有可能由于安全或性能原因而被用户有意禁止。

假如所有的样式都定义在 SVG 文件内，声明式的方法（使用标记或 CSS 属性定义的动画）在用于图像的 SVG 中依然可以运行。但动画不可交互，因为用户事件无法传递到 SVG 文档中。

目前浏览器对声明式动画方法的支持程度低于脚本动画，即便对 CSS 动画的支持程度正在增长。对于脚本动画，许多浏览器支持上的限制可以通过额外的代码来弥补（在性能上会做一些妥协），也有一些不同的代码库使得设计高效动画更加容易。

最终，当谈到动态 `fill` 和 `stroke`，渲染内容的类型会影响你对动画的选择。对于颜色，我们可以直接通过样式来修改元素上的样式属性来添加动画。但对于渲染服务，我们只能通过修改图案、渐变或结点元素上的样式和属性来添加动画。

本章中的例子将提供相同效果的多种编码方式，并突出强调每种方法的优点和局限性——包括规范上的局限和浏览器支持上的局限。

对于动画元素，以下因素应该会影响你的决定。

- 有限的浏览器支持，目前会降低我们的预期。
- 几乎能够给任何样式属性添加动画（虽然也有浏览器支持的限制）。
- 交互是通过 DOM 事件来添加，进而控制动画开始还是停止。动画会一直运行到完全结束或被其他事件终止，最终的状态也可以是“frozen”，但如果动画重新启动，它会立即恢复。
- 对于交互，正在执行动画的元素以及接收用户输入的元素可以完全独立。

- 动画可以链在一起执行或通过一个动画元素的开始或结束事件触发另一个元素来交错执行。
- 多个动画作用在同一个元素上的相同属性时，可能会一起添加（对于大多数属性），也可能后添加的动画替换前者。
- 定时动画（非交互）在 SVG 用作图像时依然可以运行。

对于 CSS 动画，需要考虑以下因素。

- 浏览器支持的限制有望改善。
- 仅受限于动画样式属性。
- 交互限制于伪类。如果伪类不再适用时动画将会中断，虽然过渡能够平滑地显示效果。
- 对于交互，添加伪类的元素必须能够影响使用 CSS 后代选择器或兄弟选择器的目标元素。这在给渲染服务添加动画时，会有很多问题。
- 链式动画会有很多困难，无论是一个 `keyframes` 序列包含了所有动画阶段，还是多个动画通过写死延迟值来匹配前一个动画持续的时间。
- 多个动画作用在同一个元素上的相同属性时，后添加的动画会替换前者。
- 定时动画（非交互）在 SVG 用作图像时依然可以运行。

最后，对于 JavaScript 动画，相关的因素如下。

- 最佳的浏览器支持，尤其是可以给旧的浏览器添加 polyfills 库。但用户有可能会禁用 JavaScript。
- 能够给任何样式、属性、文本内容、甚至 DOM 结构添加动画。
- 交互可以使用 DOM 事件传递的所有信息，并影响文档中的所有元素。
- 多级和链式动画编码困难，但 JavaScript 类库可以简化你的工作（需要用户额外下载）。
- 在作为图片使用时不能运行，外部 SVG 文件必须通过 `<object>` 嵌入。

选择通常并不唯一。特别是，你可以通过 JavaScript 来触发 CSS 动画（通过改变元素的类）或动画的元素（通过动画元素的 `beginElement()` 方法）。这使得你可以整合脚本动画的逻辑性、可控性和声明式动画的简单性。

聚焦未来

统一的 Web 动画模型

Web 动画规范提供了网页中所有定时动画的总体描述。其统一模型描述了 SVG 动画元素和 CSS 动画特性的实现。它还定义了动态创建动画效果的脚本 API。这些效果将会通过指定 `start` 和 `end` 的值以及定时参数来创建，而

不需要设置每一帧的中间值。这将结合脚本动画的可控性和声明式动画语法的简单性及性能优势。

在编写本书时，大多数浏览器都正在实现 Web 动画 API。在其他浏览器中有一些 JavaScript 的 polyfill 库，它们会把 API 指令转换为一帧一帧的动画。但 polyfill 库一般不会像其他 JavaScript 动画库一样对性能进行优化。

无论以何种方式给 SVG 添加动画，通常都是直接简单地操作填充或描边的颜色值。大多数其他表现属性也同样如此，包括 `fill-opacity` 或 `stroke-opacity`。

例 14-1 列出了动画元素的基础语法，用于循环使用不同的颜色来填充图中三颗星星。颜色循环周期是 3s，但会无限重复。每颗星星使用的颜色序列是一样的，但在不同的时间开始循环。图 14-1 显示的是循环过程中的一个瞬间。

例 14-1　使用动画元素给填充色添加动画

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="2in" viewBox="0 0 200 100">
    <title xml:lang="en">Simple SMIL Animation</title>
    <symbol id="star" viewBox="0 0 200 200">
        <path d="M100,10 L150,140 20,50 180,50 50,140 Z" />
    </symbol>
    <rect height="100%" width="100%" fill="#222"/>
    <g fill="gold">
        <use xlink:href="#star" width="50" height="50"
             transform="translate(10,20) rotate(-10)">
            <animate attributeName="fill"
                     values="gold;lightYellow;gold;tomato;gold"
                     dur="3s" repeatDur="indefinite" />
        </use>
        <use xlink:href="#star" width="40" height="40"
             transform="translate(140,10) rotate(20)">
            <animate attributeName="fill" begin="-1s"
                     values="gold;lightYellow;gold;tomato;gold"
                     dur="3s" repeatDur="indefinite" />
        </use>
        <use xlink:href="#star" width="35" height="35"
             transform="translate(80,60) rotate(-5)">
            <animate attributeName="fill" begin="-2s"
                     values="gold;lightYellow;gold;tomato;gold"
                     dur="3s" repeatDur="indefinite" />
         </use>
     </g>
</svg>
```



图 14-1：颜色变化的星星的动画序列中的一帧

颜色根据它们的 RGB 值来过渡。过渡的中间值使用和渐变相同的规则来计算。唯一的不同是动画的过渡是时间上的，而非空间上的。

颜色动画会受到 `color-interpolation` 属性的影响。但正如第 3 章中提到的，该属性在 Web 浏览器中的支持程度并不友好。

SVG 动画元素使用的是同步多媒体继承语言（SMIL）的语法，它的目的是为了整合音频、视频以及 XHTML 内容。你应该可以想象到，语法中包括多种同步多个动画的选项，控制它们同时执行还是一个接一个执行。因此，它可以用于排列复杂的动画序列。但这些复杂的动画序列不能很好地和交互特性结合在一起。SMIL 动画的声明格式中没有办法来响应当前动画状态下的用户事件。

动画元素语法的另一个限制是它很冗长。每个要添加动画的元素都需要一个与之对应的动画元素。无法给多个动画相同的图形组件快速应用变化，比如例 14-1 中的三颗星星。

广泛使用 SVG 动画元素最显著的限制是浏览器的支持程度较差。

IE 不支持动画元素。编写本书时，Chromium 团队宣布将弃用该特性。短期来看，这意味着会在使用动画语法的网页的控制台报告警告信息。长期来看，该动画可能会无法运行。

对于简单地给表现属性添加动画，CSS 动画提供了更加简洁、灵活的语法。例 14-2 中使用 CSS 动画创建了完全相同的星星闪烁的效果。一个 `class` 是用于给三颗星星应用相同的动画序列，`nth-of-type` 选择器用于改变单个星星的动画开始时间。

例 14-2　通过 CSS 关键帧给填充色添加动画

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="2in" viewBox="0 0 200 100">
    <title xml:lang="en">Simple CSS Animation</title>
    <style type="text/css">
        .star {
            fill: gold;
            animation: twinkle 3s infinite;
        }
        .star:nth-of-type(3n+2){
            animation-delay: -1s;
        }
        .star:nth-of-type(3n+3){
            animation-delay: -2s;
        }
        @keyframes twinkle {
            25% {fill: lightYellow;}
            50% {fill: gold; }
            75% {fill: tomato;}
        }
    </style>
    <symbol id="star" viewBox="0 0 200 200">
        <path d="M100,10 L150,140 20,50 180,50 50,140 Z" />
    </symbol>
    <rect height="100%" width="100%" fill="#222"/>
    <use xlink:href="#star" width="50" height="50" class="star"
         transform="translate(10,20) rotate(-10)"/>
    <use xlink:href="#star" width="40" height="40" class="star"
         transform="translate(140,10) rotate(20)"/>
    <use xlink:href="#star" width="35" height="35" class="star"
         transform="translate(80,60) rotate(-5)"/>
</svg>
```

SVG 中 CSS 动画的主要限制是只可以操作表现属性。几何属性（只能在 XML 中指定）不能通过 CSS 规则添加动画。SVG 2 规范中重新把一些布局属性定义为表现属性，但许多 SVG 特性依然无法通过 CSS 动画访问。

CSS 动画也不适用于长序列的连续动画，且只能响应有限的一组用户行为。

CSS 动画在一些还在使用中的旧版本的浏览器中不支持。但在除最新版外的所有 WebKit 浏览器中，CSS 动画需要添加 `-webkit-` 前缀，这意味着你将需要复制动画属性和关键帧规则。IE 不支持在 SVG 元素上使用 CSS 动画，但在 Edge 浏览器中可能加以支持。

要想让动画得到更普遍的支持，你可以使用 JavaScript 来操作 DOM。对于简单的线性动画，你可以直接循环所有的中间值，并使用 `requestAnimationFrame` 方法来在浏览器更新显示时更新图形。对于更复杂的多级动画（例如交错的颜色变化），你可能需要使用专门的 JavaScript 动画库，它将会把你声明的语句（哪些属性添加动画，改变多少）进行优化并一帧一帧地调整。

14.2 坐标动画

填充和描边的颜色动画是比较直接的。无论是动画元素还是 CSS 动画，浏览器都会在你指定的颜色之间过渡。但对于渲染服务是怎样的呢？这会变得更加复杂。`fill` 和 `stroke` 属性实际使用的值将会是一个 URL。浏览器无法生成两个 URL 之间的中间值。

规范中允许在动画过程中修改 URL 的值。但浏览器的支持程度较差，编写本书时，仅限于 Firefox 的 SMIL 动画。

但渲染服务元素本身是可以添加动画的。渐变结点的颜色和不透明度的值可以通过 CSS 添加动画，也可以对图案内容的表现样式添加动画。在将来，渐变和图案的转换也会受到 CSS 动画的 `transform` 属性影响。这些属性以及更多的结构属性可以通过 SVG 标记或 JavaScript 来添加动画。

给渲染服务添加动画与给 `fill` 和 `stroke` 渲染属性添加动画有一个很重要的不同。动画会自动影响所有使用该渲染服务的元素，且同时改变它们。

这个特性是否真的需要取决于图形的具体细节。14.3 节将探讨当你不希望所有元素都同时改变时的代替方法。特别之处是该部分将重点关注使用动画来突出用户正在进行交互的特定元素。相比之下，对于许多声明式动画来说，同步动画是可接受的，甚至是我们所期望的。

例 14-3 中使用同步动画来改编星星闪烁的动画。现在，我们不是给星星上的填充颜色添加动画，而是把动画应用在一系列 `<stop>` 元素的 `stop-color` 值。动画代码看起来非常相似。图 14-2 显示了渐变填充的星星的一个屏幕截图，你需要自己运行代码来看到完整的闪烁效果。

例 14-3　使用 CSS 给渐变结点颜色添加动画

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="2in" viewBox="0 0 200 100">
    <title xml:lang="en">CSS-Animated Gradient</title>
    <style type="text/css">
        .star {
            fill: url(#shine);
        }
        #shine stop {
            stop-color: gold;
            animation: twinkle 3s infinite;
        }
        #shine stop:nth-of-type(3n+2){
            animation-delay: -1s;
        }
        #shine stop:nth-of-type(3n+3){
            animation-delay: -2s;
        }
        @keyframes twinkle {
            25% {stop-color: lightYellow;}
            50% {stop-color: gold; }
            75% {stop-color: tomato;}
        }
    </style>
    <symbol id="star" viewBox="0 0 200 200">
        <path d="M100,10 L150,140 20,50 180,50 50,140 Z" />
    </symbol>
    <linearGradient id="shine" gradientTransform="rotate(20)">
        <stop offset="0" />
        <stop offset="0.25" />
        <stop offset="0.5" />
        <stop offset="0.75" />
        <stop offset="1" />
    </linearGradient>
    <rect height="100%" width="100%" fill="#222"/>
    <use xlink:href="#star" width="50" height="50" class="star"
         transform="translate(10,20) rotate(-10)"/>
    <use xlink:href="#star" width="40" height="40" class="star"
         transform="translate(140,10) rotate(20)"/>
    <use xlink:href="#star" width="35" height="35" class="star"
         transform="translate(80,60) rotate(-5)"/>
</svg>
```



图 14-2：使用震荡渐变的星星的动画序列的一帧

往复的颜色结点也可以使用 SMIL 式的动画元素来创建，通过在每个 <stop> 元素内添加一个 <animate> 元素。但在一些 CSS 动画无法做到的事情上，动画元素就显得更加有用：比如修改几何属性。

例 14-4 中使用两个动画元素来修改 <linearGradient> 元素的 x1 和 y2 属性。它的静态屏幕截图与图 14-2 非常相似，但动态效果会有很大不同：每个星星不只是在恰当的位置闪烁，还会像柔和的波浪一样延伸和移动。

例 14-4　使用动画元素给渐变矢量添加动画

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="2in" viewBox="0 0 200 100">
    <title xml:lang="en">SMIL-Animated Gradient</title>
    <style type="text/css">
        .star {
            fill: url(#shine);
        }
    </style>
    <symbol id="star" viewBox="0 0 200 200">
        <path d="M100,10 L150,140 20,50 180,50 50,140 Z" />
    </symbol>
    <linearGradient id="shine" spreadMethod="repeat"
                    gradientTransform="rotate(20)" >
        <animate attributeName="y2" values="1;1.5;1;0.75;1"
                 dur="2s" repeatDur="indefinite" />
        <animate attributeName="x1" values="0;0.5;0"
                 dur="3s" repeatDur="indefinite" />
        <stop offset="0" stop-color="gold" />
        <stop offset="0.25" stop-color="lightYellow" />
```

```
        <stop offset="0.5" stop-color="gold" />
        <stop offset="0.75" stop-color="tomato" />
        <stop offset="1" stop-color="gold" />
    </linearGradient>
    <rect height="100%" width="100%" fill="#222"/>
    <use xlink:href="#star" width="50" height="50" class="star"
         transform="translate(10,20) rotate(-10)"/>
    <use xlink:href="#star" width="40" height="40" class="star"
         transform="translate(140,10) rotate(20)"/>
    <use xlink:href="#star" width="35" height="35" class="star"
         transform="translate(80,60) rotate(-5)"/>
</svg>
```

当渐变使用 SMIL 添加动画时，Firefox 和 WebKit 浏览器目前都不会更新绘制的图形（只能在 Blink 浏览器中工作）。

使用 JavaScript（而不用 SMIL）修改渐变属性来创建这种效果可以获得更好的浏览器支持。但这里三颗星星的动画将完全同步，因为他们都使用相同的渐变。

对于某些图形，也许你可以通过使用每个形状内略不相同的重复 `userSpaceOnUse` 图形来扰乱这种重复效果。但对于这三颗星星，这种方式没有任何效果：每颗星星相对于自己的 `<symbol>` 坐标系位置相同，所以每颗星星绘制在用户空间渐变内相同的点上。

聚焦未来
渐变之间的过渡

依据 CSS Image Values and Replaced Content Module Level 3 的最新草案，在一些浏览器中可以给 CSS 渐变函数添加动画（通过 CSS 动画）。要想在两个渐变之间过渡，它们的类型必须相同（例如，都是 `repeating-linear-gradient`），且结点的数量也必须相同。之后单独在结点偏移和颜色上进行过渡。

不支持渐变插值的浏览器将会忽略所有在 `@keyframes` 中设置的值，且不进行过渡并立即应用其他的变化。

CSS 过渡规范中有提案建议通过 URL 引用 SVG 渐变可以以同样的方式过渡。换句话说，如果两个渐变类型相同且结点的个数也相同，当通过 CSS

动画或过渡切换渲染服务时，浏览器将会生成所有的中间值。编写本书时，还没有任何浏览器实现该功能，且对把它加入最终规范存在反对意见。Level 4 规范中引入了一个替代的过渡模式 cross-fading，它将允许任何图片溶解到另一张图片内，这更方便应用在 SVG 渲染服务的内容中。

14.3 交互动画

动画在现代站点中最重要的用途之一是提供用户反馈和连续性。在用户进行交互时，内容应该明显变化，但变化应该平滑过渡，这样用户才可以直观地理解新旧状态的关系。

如果使用动画来表示用户和特定元素之间的交互，你显然不能让网页上所有相似的元素同时触发相同的动画。一种解决方式是专门创建一个渲染服务元素（或多个元素）来应用动画效果，且只在需要的时候使用它。同时通过其他渲染服务绘制元素的静态状态。

当用户和一个元素（图形图标）交互时，给另一个元素（渲染服务）添加动画难以通过 CSS 动画实现。有时可以通过兄弟选择器和子选择器来完成，但你必须把标记重新排列为一个相当凌乱的结构。因此，我们首先使用基于 SMIL 的方法来描述要实现什么，然后展示如何使用 JavaScript（工作量更大，但更加灵活且浏览器支持较好）创建相同的效果。

SMIL 动画方法利用定时属性来同步多个动画。使用 <set> 元素来把共享的、静态的渐变渲染服务切换到有动画的渲染服务，然后使用 <animate> 元素来实现效果。最后，另一个 <set> 元素把图形切换到最终状态，即再次变为共享的渲染服务。

> Blink 和 WebKit 浏览器不会正确地给应用渲染服务的 URL 添加动画，即使使用 <set> 元素也不行，填充值将会被透明（WebKit）或纯黑色（Blink）替换。由于 Chromium 项目计划不推荐使用 SMIL，因此预计这不会被修复。所以该例子只能在 Firefox 中运行。

例 14-5 中使用这种方法实现了星星图标上的点击效果，这可能会用在喜欢或书签元素上。它使用我们之前在第 8 章星级图标示例中看到的银色和金色线性渐变，以及一个额外的径向渐变来作为过渡效果。

星星刚开始使用银 - 灰渐变来填充。激活后，在变为金色线性渐变之前，一个金色的填充从元素的中心辐射出来。动画效果设计所有三个渐变，在一个 `<pattern>` 元素内布局。除了给径向渐变的半径添加动画外，图层的不透明度也在平滑地过渡。

图 14-3 显示了一个平面截图，其中一个图标已经是最终的金色状态，一个是初始的银色状态，另一个刚刚被点击且辐射动画只进行了一部分。



图 14-3：一个交互动画网页的截图，最后一个图标刚被选中不久

例 14-5　通过动画元素给单个元素添加响应用户交互的动画

HTML 标记：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>Interactive Animated Gradient with SMIL</title>
    <style type='text/css'>
        /* styles must be in the same document */
    </style>
</head>
<body>
    <svg class="defs-only"
         aria-hidden="true" focusable="false" width="0" height="0" >
        <linearGradient id="silver-shine" spreadMethod="repeat"
                        gradientTransform="rotate(20)" >
            <stop offset="0" stop-color="gray" />
```

```
        <stop offset="0.35" stop-color="silver" />
        <stop offset="1" stop-color="gray" />
    </linearGradient>
    <linearGradient id="gold-shine" spreadMethod="repeat"
                    gradientTransform="rotate(20)" >
        <stop offset="0" stop-color="gold" />
        <stop offset="0.35" stop-color="lightYellow" />
        <stop offset="1" stop-color="gold" />
    </linearGradient>
    <radialGradient id="gold-ripple" r="0.2">                 ❶
        <animate id="ripple"
                 attributeName="r" from="0.1" to="1"
                 dur="0.7s" fill="freeze"
                 begin="reaction.begin + 0.1s" />
        <stop offset="0" stop-color="gold" />
        <stop offset="0.5" stop-color="lightYellow" />
        <stop offset="0.75" stop-color="silver" />
        <stop offset="1" stop-color="gray" />
    </radialGradient>
    <pattern id="turn-gold" width="1" height="1"
             patternContentUnits="objectBoundingBox">         ❷
        <rect fill="url(#gold-shine)" width="1" height="1" />
        <rect fill="url(#silver-shine)" width="1" height="1" >
            <set attributeName="opacity" to="0"
                 begin="reaction.begin + 0.5s" />             ❸
        </rect>
        <rect fill="url(#gold-ripple)" width="1" height="1" >
            <animate id="reaction"
                     attributeName="opacity"
                     values="0;1;1;0"
                     keyTimes="0;0.2;0.8;1"
                     dur="1s"
                     begin="switch1.begin;
                     switch2.begin; switch3.begin" />
        </rect>
    </pattern>

    <symbol id="star" viewBox="0 0 200 200">
        <path d="M100,10 L150,140 20,50 180,50 50,140 Z" />
    </symbol>
</svg>
<h1>Pick your favorites:</h1>
<div>
    <figure id="opt1" role="checkbox" tabindex="0">           ❹
        <figcaption>Option 1</figcaption>
        <svg><use class="star" xlink:href="#star">
            <set id="switch1"
                 attributeName="fill" to="url(#turn-gold)"
                 dur="1s" begin="opt1.click; opt1.activate"/>
```

```
                <set attributeName="fill" to="url(#gold-shine)"
                     dur="indefinite" begin="switch1.end" />    ❺
            </use></svg>
        </figure>
        <figure id="opt2" role="checkbox" tabindex="0">
            <figcaption>Option 2</figcaption>
            <svg><use class="star" xlink:href="#star">
                <set id="switch2"
                     attributeName="fill" to="url(#turn-gold)"
                     dur="1s" begin="opt2.click; opt2.activate"/>
                <set attributeName="fill" to="url(#gold-shine)"
                     dur="indefinite" begin="switch2.end" />    ❻
            </use></svg>
        </figure>
        <figure id="opt3" role="checkbox" tabindex="0">
            <figcaption>Option 3</figcaption>
            <svg><use class="star" xlink:href="#star">
                <set id="switch3"
                     attributeName="fill" to="url(#turn-gold)"
                     dur="1s" begin="opt3.click; opt3.activate"/>
                <set attributeName="fill" to="url(#gold-shine)"
                     dur="indefinite" begin="switch3.end" />
            </use></svg>
        </figure>
    </div>
    <script>
        /* do something based on the selected options */
    </script>
</body>
</html>
```

❶ 前两个是普通的线性渐变。但新的径向渐变包含一个 <animate> 元素来操作它的 r 属性。该动画在 id 为 reaction 的动画开始执行十分之一秒后执行。

❷ <pattern> 元素包含了所有需要添加过渡动画的层，作为单个磁贴来完全填充形状边界盒。

❸ 图案内还嵌套了两个动画元素：<set> 元素把位于下面的层从银色切换到金色，<animate> 元素对径向渐变的不透明度先增后减。后一个动画有一个 reaction 的 ID 值，用于触发其他两个过渡动画。reaction 动画本身在三个切换动画中任一个开始的时候启动。

❹ 在主要的网页标记中，包裹 SVG 动画的 <figure> 元素等价于复选框，所以给它添加了 ARIA 的 role="checkbox" 标识。每个 SVG 包含一个 <use> 元素，和内联图标一样。但每个 <use> 元素内还包含两个动画元素。

❺ switch1 动画是一个响应复选框 <figure> 上点击或激活事件的 <set> 元素，它会将相关图标从银色渐变切换到过渡图案，然后开始执行动画。第二个 <set> 元素在动画完成后再次把填充切换为最终的金色状态。

❻ 其他的图标使用相同的动画结构，但是 ID 都唯一。

CSS 样式：

```
html {
    background-color: #222;
    color: lightSkyBlue;
}
svg.defs-only {
    display: block;
    position: absolute;
    height: 0; width: 0;
    overflow: hidden;
}

figure[role="checkbox"] {
    display: inline-block;
    max-width: 33%;
    min-width: 5em;
    padding: 0; margin: 0;
    font-size: larger;                         ❶
}
figcaption {
    display: block;
    text-align: center;
}
figure[role="checkbox"] > svg {
    display: block;
    margin: auto;
    width: 4em;
    height: 4em;
}
.star {
    fill: url(#silver-shine);
    cursor: pointer;                           ❷
}
```

❶ 使用 role 属性值为 checkbox 来控制布局，而没有用 CSS 类选择器。

❷ CSS 中同样设置星星图标的默认填充值为银灰色渐变。将鼠标指针设为 pointer 将使鼠标用户知道该内容是可交互的。

除了浏览器支持程度低以外，该方法还有其他局限。

- 如果用户在 1 秒内选择了两个不同的图标，第一个动画将会被重置为与第二个动画同步。
- 驱动主要 `reaction` 动画的动画元素必须在 `begin` 属性内添加所有可以触发它的事件。例 14-5 中通过响应 `<set>` 动画来简化它，而不是使用原始的用户事件。但如果要向网页内添加新的交互图标，仍然需要在 `begin` 属性内添加新的值。
- 在图标再一次被点击的时候，选择状态不会切换，且动画也不会回退。后续的点击会重新启动相同的动画序列，但不会取消已经选择的条目。如果想实现一个反向的动画，需要单独添加一个透明的 SVG 元素（使用另一个 `<set>` 动画），当图标被选中时显示在图标的最上层并捕获所有的用户事件（使用 `pointer-events:all`）。该元素将用于触发取消选择[2]。你还需要一个 `<animate>` 和 `<set>` 元素来控制反向的动画序列。
- 虽然图标已经赋予了 ARIA 的复选框角色，以便屏幕阅读器用户可以激活它们，但元素的选中状态在图标被选中时不会更新。这需要在脚本中实现，SMIL 动画在任何浏览器中都不可以修改 HTML 元素上的属性。

这些问题在例 14-6 中得到了解决：使用 JavaScript 来控制有复选框逻辑的动画。它还添加了对键盘的支持并增加了一个新功能：当图标通过鼠标 `click` 或 `tap` 激活时，径向渐变从触摸点开始向外扩展。这种对用户事件微妙的反应是许多交互设计指南中所推荐的，包括谷歌的 Material Design 设计指南。

例 14-6　通过 JavaScript 给单个元素添加响应用户交互的动画

HTML 标记：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>Interactive Animated Gradient with JavaScript</title>
    <style type='text/css'>
        /* styles unchanged */                                        ❶
    </style>
</head>
<body>
```

注 2：如果你对创建切换的 SMIL 动画感兴趣（对于其他类型的动画，比如形状变形，浏览器的支持程度更好），Michael J. Harris 写了一个很好的教程（http://codepen.io/mikemjharris/blog/svg-toggling）。

```
<svg class="defs-only"
     aria-hidden="true" focusable="false" width="0" height="0" >
    <linearGradient id="silver-shine" spreadMethod="repeat"
                    gradientTransform="rotate(20)" >
        <stop offset="0" stop-color="gray" />
        <stop offset="0.25" stop-color="silver" />
        <stop offset="1" stop-color="gray" />
    </linearGradient>
    <linearGradient id="gold-shine" spreadMethod="repeat"
                    gradientTransform="rotate(20)" >
        <stop offset="0" stop-color="gold" />
        <stop offset="0.25" stop-color="lightYellow" />
        <stop offset="1" stop-color="gold" />
    </linearGradient>
    <radialGradient id="gold-ripple" r="0.2">
        <stop offset="0" stop-color="gold" />
        <stop offset="0.5" stop-color="lightYellow" />
        <stop offset="0.75" stop-color="silver" />
        <stop offset="1" stop-color="gray" />
    </radialGradient>
    <pattern id="turn-gold" width="1" height="1"
             patternContentUnits="objectBoundingBox">        ❷
        <rect class="transition"
              fill="url(#gold-ripple)" width="1" height="1" />
        <rect class="on"
              fill="url(#gold-shine)" width="1" height="1" />
        <rect class="off"
              fill="url(#silver-shine)" width="1" height="1" />
    </pattern>
    <symbol id="star" viewBox="0 0 200 200">
        <path d="M100,10 L150,140 20,50 180,50 50,140 Z" />
    </symbol>
</svg>
<h1>Pick your favorite:</h1>
<div>
    <figure id="opt1" role="checkbox">
        <figcaption>Option 1</figcaption>
        <svg><use xlink:href="#star"/></svg>                  ❸
    </figure>
    <figure id="opt2" role="checkbox">
        <figcaption>Option 2</figcaption>
        <svg><use xlink:href="#star"/></svg>
    </figure>
    <figure id="opt3" role="checkbox">
        <figcaption>Option 3</figcaption>
        <svg><use xlink:href="#star"/></svg>
    </figure>
</div>
```

```
    <script>
        /* Script included at end of file
           or as an async-loaded external resource */
    </script>
</body>
</html>
```

❶ 基础的样式与例 14-5 中完全一样。

❷ 渐变和图案基本一样。删除了动画元素，且图案层级重新排列以使动画代码更加简单。

❸ 除删除了动画元素外，网页标记也相同。

JavaScript：

```
(function(){
    var toggles = document.querySelectorAll("[role='checkbox']");
    var selectGraphic = ".star";
    var svg = document.querySelector("svg");
        //arbitrary <svg> element so that
        //we can access SVG dom methods
    var paint = {
                off: "url(#silver-shine)",
                animate: "url(#turn-gold)",
                on: "url(#gold-shine)"
            };                                              ❶
    var animating = false,
        animatingOption,
        nextFrame;
    for (var i=0, n=toggles.length; i<n; i++){              ❷
        toggles[i].setAttribute("aria-checked", false);
        toggles[i].querySelector(selectGraphic).style.fill
            = paint.off;
        toggles[i].addEventListener("click", toggleState);
        toggles[i].addEventListener("keyup", checkKey);

        //tell Internet Explorer not to focus <svg>
        toggles[i].querySelector("svg")
                  .setAttribute("focusable", false);
    }
    function checkKey(e) {                                  ❸
        //check for spacebar or Enter key,
        //using both the new standard syntax
        //and the old keycode syntax
        if ( (e.key == " ")||(e.key == "Enter") ||
             (e.keyCode == 32 )||(e.keyCode == 13) )
            toggleState.apply(this, arguments);
```

```
}
function toggleState(e){                                              ❹
    var currentlyChecked =
            (this.getAttribute("aria-checked")
                    === true.toString() );

   //update the actual state
    this.setAttribute("aria-checked", !currentlyChecked);
        /* maybe do something based on the selected options */

    if (currentlyChecked) {
        //animate turning off quickly
        animateStar(this, e, 300, true);                              ❺
    }
    else {
        //animate turning on, more slowly
        animateStar(this, e, 1000);
    }
}

function animateStar(option, event, dur, reverse) {
    if ((!dur)||isNaN(dur)) return;
        //must have a valid animation duration

    /* animation parameters */
    var effects = [                                                   ❻
                {selector:"#gold-ripple", attr:"r",
                 from:0.1, to:1, t1:0.1, t2:0.8},
                {selector:"#turn-gold .on", attr:"opacity",
                 from:0, to:1, t1:0.8, t2:1},
                {selector:"#turn-gold .off", attr:"opacity",
                from:1, to:0, t1:0, t2:0.2}
                ];
    var selectTracker = "#gold-ripple";
    var star = option.querySelector(selectGraphic);
    var startTime = 0;

    if (reverse) {
        //swap the order of each effect
        effects.forEach(function(effect){
            var swap = effect.from;
            effect.from = effect.to;
            effect.to = swap;

            swap = effect.t1;
            effect.t1 = 1 - effect.t2;
            effect.t2 = 1 - swap;
        });
```

```
    }

    if (animating){                                                    ❼
        //abort current animation
        cancelAnimationFrame(nextFrame);

        if (animatingOption != option) {
            //tidy up the current animation by setting
            //it to the correct end state
            animatingOption.querySelector(selectGraphic)
                .style.fill =
                (animatingOption.getAttribute("aria-checked")
                    === true.toString() ) ?
                paint.on : paint.off;
        }
        else {
            //set the new animation to start
            //from the current state
            effects.forEach(function(effect){
                effect.from = parseFloat(
                    document.querySelector(effect.selector)
                            .getAttribute(effect.attr) );
             });
        }
    }

    var track = document.querySelector(selectTracker);
    if (event instanceof MouseEvent) {                                 ❽
        //recenter the radial gradient
        //to track the mouse event
        //by converting mouse coordinates first to
        //userSpace coordinates for the star,
        //and then to bounding box coordinates
        var bbox = star.getBBox(),
            CTM = star.getScreenCTM().inverse(),
            p = svg.createSVGPoint(),
            p2, newCx, newCy;
        p.x = event.clientX; //NOT screenX and screenY
        p.y = event.clientY;
        p2 = p.matrixTransform(CTM);
        newCx = (p2.x - bbox.x)/bbox.width;
        newCy = (p2.y - bbox.y)/bbox.height;
        if (!animating ||(animatingOption != option)) {
             //start immediately at the mouse point
             track.setAttribute("cx", newCx);
             track.setAttribute("cy", newCy);
         }
         else {
```

```
        //create an animated shift in the gradient center
        var oldCx = parseFloat(track.getAttribute("cx"));
        var oldCy = parseFloat(track.getAttribute("cy"));
        effects.push(
             {selector: selectTracker, attr:"cx",
              from: oldCx, to: newCx,
              t1: 0, t2: 0.2
            } );
        effects.push(
            {selector: selectTracker, attr:"cy",
             from: oldCy, to: newCy,
             t1: 0, t2: 0.2
            } );
    }
}
else {
     //center gradient if triggered by keyboard event
     track.setAttribute("cx", 0.5);
     track.setAttribute("cy", 0.5);
}

//set overall animation parameters and initialize
animating = true;
animatingOption = option;
requestAnimationFrame(function(t){
     startTime = t;
     star.style.fill = paint.animate;                    ❾
});

//create a function to transform a time point
//into a position in the animation effects
var getProgress = function(t){ return (t-startTime)/dur; };

//determine which element will be animated for each effect
//and the total amount of change
effects.forEach(function(effect){
    effect.node = document.querySelector(effect.selector);
    effect.by = effect.to - effect.from;
});

function applyEffects(t){                                ❿
    var a = getProgress(t),
        val;
    effects.forEach(function(effect){
        if(a <= effect.t1) {
            val = effect.from;
        }
        else if (a >= effect.t2) {
```

```
                val = effect.to;
            }
            else {
                val = effect.from + effect.by*(
                    (a - effect.t1)/(effect.t2 - effect.t1) );
            }
            effect.node.setAttribute(effect.attr, val);
        });

        if ( a < 1) {
            //loop
            nextFrame = requestAnimationFrame(applyEffects);
        }
        else {
            //animation is complete
            star.style.fill = reverse? paint.off : paint.on;
            animating = false;
        }
    }
    //start updating
    nextFrame = requestAnimationFrame(applyEffects);
  }
})();
```

❶ 脚本开始先声明一组函数调用期间共用的常量和变量。包括一组类复选框的元素，用于标识将要更改的图形的选择器，一个随机的将在 SVG 通用方法内使用的 `<svg>` 元素，以及将会使用的特定填充值的引用，这样我们就可以方便地更新他们。

❷ for 循环初始化每个切换元素（类复选框元素）为未选中的状态，同时添加事件监听器。

❸ `checkKey` 辅助函数用于响应键盘事件，并检测切换的状态是否应该改变。如果符合要求，则调用主要的事件处理函数并传入当前上下文以及参数。

❹ `toggleState` 方法可以直接通过点击事件触发或间接通过键盘事件触发。它会更改当前元素的 `aria-checked` 属性，并开始过渡动画。

❺ `animateStar` 函数进行开始动画前需要的所有计算。参数表示动画应该持续多长时间，动画是否是从选中状态回退到未选中状态。

❻ 整个动画定义为一个不同效果的数组。每种效果的对象描述了要修改哪个元素（使用 CSS 选择器的值），改变哪个属性（`attr`），初始值（`from`）和最终值（`to`）分别是什么，以及动画的开始（`t1`）和结束

（t2）点与整个动画持续时间的比例。硬编码的值描述的是动画的方向，如果动画应该反向运行，就会交换对应的值。

❼ if(animating) 代码块包含了防止两个动画冲突的代码，它们修改的是同一个渲染服务元素。

❽ 接下来的代码块首先测试动画是否是鼠标点击屏幕触发的，如果是，就调整渐变围绕触摸点向外辐射或向内收缩。

❾ 只调用一次的 requestAnimationFrame 函数用于保存动画开始的时间戳，并把渲染服务变为有动画的图案。

❿ 动画的剩余部分使用 requestAnimationFrame 重复调用 applyEffects 函数，以此来更新所有过渡属性而不必担心大量额外的计算影响它的速度。

标记几乎与例 14-5 中完全相同，只是删除了动画元素。唯一的改变是 <pattern> 元素内层级的顺序——这个改变允许每个动画效果使用从一个值变为另一个值的单向动画来描述。我们没有让有动画的径向渐变层先淡入后淡出，而是把它绘制为最下面的一层，然后让上面的两个线性渐变层淡入或淡出。

触发用户事件后总体操作如下。

(1) 确定该操作是选中还是取消，并更新 ARIA 的状态（toggleState 函数）。这里也是添加所有与选中和取消选中相关的行为的地方。

(2) 调用动画启动函数（animateStar），运行 1000ms 的开启动画或 300ms 的关闭动画。时间以毫秒为单位，这是因为大多数 JavaScript 和 DOM 的时间方法使用的是毫秒。关闭动画运行得更快，因为我们想要非常明确地响应用户取消选择的操作。

(3) 检查是否有动画正在执行，如果有，就取消挂起的动画请求。如果正在执行动画的是另一个星星，立即把该星星置为最终状态。另外，如果是同一颗星星，调整当前动画的参数使它可以从当前状态平稳地继续。

(4) 如果是鼠标事件（包括触摸屏上 tap 创建的点击），把径向渐变的中心设为该触发事件的点。本节还使用了许多 SVG DOM 特有的方法。

- element.getBBox() 返回对象在其局部坐标系中的边界盒，值为包含 x、y、width 和 height 属性的对象。
- element.getScreenCTM() 返回元素坐标系与页面根坐标系之间的累积变换矩阵（CTM）。它不是鼠标事件使用的屏幕坐标系，而是客户端坐标系。

- `matrix.inverse()` 返回一个新的变换矩阵，完全逆转原矩阵创建的变换。
- `svg.createSVGPoint()` 生成保存 x 和 y 数值的 SVG 数据对象。SVGPoint 对象在访问多边形或折线时使用。自己创建的唯一原因是用于下一个方法。
- `point.matrixTransform(matrix)` 应用特定矩阵变换来计算给定的点的位置。

通过这些方法，鼠标事件的坐标可以转换到星星使用的坐标系，然后使用星星的边界盒来把它转换为渐变的 `cx` 和 `cy` 属性的对象边界盒坐标。如果更改的是正在执行的动画，中心点的移动将通过给 `effects` 数组添加新的对象来应用其他动画效果。相反，如果切换是由键盘事件触发，`cx` 和 `cy` 将被重置为元素的中心。

(5) 调用一个只执行一次的动画函数来初始化动画，修改星星的填充值为动画图案，并记录动画循环的开始时间。

(6) 定义一个函数（`getProgress`）将文档的时间戳转化为动画执行时间与动画周期的比。如果你想通过缓动函数来减慢或加快动画的进度，这里就是进行操作的地方。

(7) 标识每个动画对应的元素，并把它存储在变量中以便快速访问。同时还把 `from-to` 语法转化为 `from-by` 的格式以便动画的每一帧中的数学计算更快。

(8) 最后，根据浏览器响应动画帧请求时传递给回调函数（`applyEffects`）的时间戳，在动画的每个阶段应用变化效果。效果是通过数据对象数组来定义的，每个对象的结构都是一样的，所以可以通过单个函数来计算当前值以及更新特定元素上相应的属性。

这里使用的方法仅仅支持简单的数值插入，且只支持更改属性（包括表现属性）。如果想要给颜色、复杂的数据列表或动画样式值来添加动画，需要添加额外的代码。

(9) 在动画的回调中，会检查动画是否完成。如果还没有完成，就请求另一个动画帧。如果已经完成，把星星切换到最终渐变状态。

唯一的限制是如果另一颗星星上有交互，当前星星的动画会被中止。这是避免冲突所必需的，因为这里只有一个渲染服务来绘制所有星星的动画状态。为了解决这个问题，你需要动态为每个要添加动画的元素单独创建渲染服务（`<radialGradient>` 以及引用它的 `<pattern>`）的副本。这些 DOM

操作会影响性能，所以你要决定对于小概率的交互重叠是否值得这样做。当然，两个交互重叠的可能性取决于每个动画持续多长时间。对于用户交互动画，1 秒的动画实际上被认为是相当长的动画。

一个微小的交互动画就需要这么多的代码。因此，很少会为每个项目编写完整的自定义动画脚本。JavaScript 库，例如 Snap.svg（*http://snapsvg.io/*）、D3.js（*https://d3js.org/*）、GreenSock Animation Platform 和 GSAP（*https://greensock.com/*）等，提供了有效控制动画时间和中间插值的方法。随着浏览器对 Web 动画 API 支持程度的增加，这将成为定义特定动画效果的首选方法。

但通过切换渲染服务来创建动画和给渲染服务组件添加动画的整体策略在使用这些工具时依然适用。也就是说，当浏览器对分层填充、使用 CSS 渐变填充以及 CSS 渐变函数的过渡的支持程度更高时，这种特殊效果将会变得更加容易。

附录A

颜色关键词和语法

SVG 和 CSS 中的颜色可以指定为精确的数值，也可以使用预定义的颜色关键词。

自定义纯色可以通过以下任意一种格式来指定。

- `#RRGGBB`，六个十六进制的数字，每一对代表一种颜色通道在 0（00）到 255（FF）之间的一个值。
- `#RGB`，三个十六进制的数字，相当于每个数字出现两次的六位版本。
- `rgb(r,g,b)`，三个 0 到 255 之间的整数或三个百分数值。
- `hsl(h,s,l)`，第一个参数（hue）是一个表示色轮（相对于红色）上角度的数值，饱和度和亮度值是百分数。

自定义部分透明颜色可以通过以下两个函数之一来指定。

- `rgba(r,g,b,a)`，前三个值是百分数或整数，`a` 的值是 0 到 1 之间的小数。
- `hsla(h,s,l,a)`，参数值是一个数值，两个百分数，之后是 0 到 1 之间的 alpha 值。

命名纯色在表 A-1 中列出，同时还给出了与之等价的十六进制、RGB 和 HSL 格式。百分比的 RGB 值和 HSL 值已经四舍五入为整数。此外，颜色关键词 `transparent` 表示为 `rgba(0,0,0,0)` 或 `hsla(0,0%,0%,0)`。

表A-1　SVG和CSS中的命名颜色

关键字	十六进制值	RGB 十进制值			RGB 百分比			HSL		
		R	G	B	R	G	B	H	S	L
AliceBlue	#f0f8ff	240	248	255	94%	97%	100%	208	6%	97%
antiqueWhite	#faebd7	250	235	215	98%	92%	84%	34	14%	91%
aqua	#00ffff	0	255	255	0%	100%	100%	180	100%	50%
aquamarine	#7fffd4	127	255	212	50%	100%	83%	160	50%	75%
azure	#f0ffff	240	255	255	94%	100%	100%	180	6%	97%
beige	#f5f5dc	245	245	220	96%	96%	86%	60	10%	91%
bisque	#ffe4c4	255	228	196	100%	89%	77%	33	23%	88%
black	#000000	0	0	0	0%	0%	0%	0	0%	0%
blanchedAlmond	#ffebcd	255	235	205	100%	92%	80%	36	20%	90%
blue	#0000ff	0	0	255	0%	0%	100%	240	100%	50%
blueViolet	#8a2be2	138	43	226	54%	17%	89%	271	81%	53%
brown	#a52a2a	165	42	42	65%	16%	16%	0	75%	41%
burlywood	#deb887	222	184	135	87%	72%	53%	34	39%	70%
cadetBlue	#5f9ea0	95	158	160	37%	62%	63%	182	41%	50%
chartreuse	#7fff00	127	255	0	50%	100%	0%	90	100%	50%
chocolate	#d2691e	210	105	30	82%	41%	12%	25	86%	47%
coral	#ff7f50	255	127	80	100%	50%	31%	16	69%	66%
cornflowerBlue	#6495ed	100	149	237	39%	58%	93%	219	58%	66%
cornsilk	#fff8dc	255	248	220	100%	97%	86%	48	14%	93%
crimson	#dc143c	220	20	60	86%	8%	24%	348	91%	47%
cyan	#00ffff	0	255	255	0%	100%	100%	180	100%	50%
darkBlue	#00008b	0	0	139	0%	0%	55%	240	100%	27%
darkCyan	#008b8b	0	139	139	0%	55%	55%	180	100%	27%
darkGoldenrod	#b8860b	184	134	11	72%	53%	4%	43	94%	38%
darkGray	#a9a9a9	169	169	169	66%	66%	66%	0	0%	66%
darkGreen	#006400	0	100	0	0%	39%	0%	120	100%	20%
darkGrey	#a9a9a9	169	169	169	66%	66%	66%	0	0%	66%
darkKhaki	#bdb76b	189	183	107	74%	72%	42%	56	43%	58%
darkMagenta	#8b008b	139	0	139	55%	0%	55%	300	100%	27%
darkOliveGreen	#556b2f	85	107	47	33%	42%	18%	82	56%	30%
darkOrange	#ff8c00	255	140	0	100%	55%	0%	33	100%	50%

（续）

关键字	十六进制值	RGB 十进制值			RGB 百分比			HSL		
		R	G	B	R	G	B	H	S	L
darkOrchid	#9932cc	153	50	204	60%	20%	80%	280	75%	50%
darkRed	#8b0000	139	0	0	55%	0%	0%	0	100%	27%
darkSalmon	#e9967a	233	150	122	91%	59%	48%	15	48%	70%
darkSeaGreen	#8fbc8f	143	188	143	56%	74%	56%	120	24%	65%
darkSlateBlue	#483d8b	72	61	139	28%	24%	55%	248	56%	39%
darkSlateGray	#2f4f4f	47	79	79	18%	31%	31%	180	41%	25%
darkSlateGrey	#2f4f4f	47	79	79	18%	31%	31%	180	41%	25%
darkTurquoise	#00ced1	0	206	209	0%	81%	82%	181	100%	41%
darkViolet	#9400d3	148	0	211	58%	0%	83%	282	100%	41%
deepPink	#ff1493	255	20	147	100%	8%	58%	328	92%	54%
deepSkyBlue	#00bfff	0	191	255	0%	75%	100%	195	100%	50%
dimGray	#696969	105	105	105	41%	41%	41%	0	0%	41%
dimGrey	#696969	105	105	105	41%	41%	41%	0	0%	41%
dodgerBlue	#1e90ff	30	144	255	12%	56%	100%	210	88%	56%
firebrick	#b22222	178	34	34	70%	13%	13%	0	81%	42%
floralWhite	#fffaf0	255	250	240	100%	98%	94%	40	6%	97%
forestGreen	#228b22	34	139	34	13%	55%	13%	120	76%	34%
fuchsia	#ff00ff	255	0	255	100%	0%	100%	300	100%	50%
gainsboro	#dcdcdc	220	220	220	86%	86%	86%	0	0%	86%
ghostWhite	#f8f8ff	248	248	255	97%	97%	100%	240	3%	99%
gold	#ffd700	255	215	0	100%	84%	0%	51	100%	50%
goldenrod	#daa520	218	165	32	85%	65%	13%	43	85%	49%
gray	#808080	128	128	128	50%	50%	50%	0	0%	50%
green	#008000	0	128	0	0%	50%	0%	120	100%	25%
greenYellow	#adff2f	173	255	47	68%	100%	18%	84	82%	59%
grey	#808080	128	128	128	50%	50%	50%	0	0%	50%
honeydew	#f0fff0	240	255	240	94%	100%	94%	120	6%	97%
hotPink	#ff69b4	255	105	180	100%	41%	71%	330	59%	71%
indianRed	#cd5c5c	205	92	92	80%	36%	36%	0	55%	58%
indigo	#4b0082	75	0	130	29%	0%	51%	275	100%	25%
ivory	#fffff0	255	255	240	100%	100%	94%	60	6%	97%

（续）

关键字	十六进制值	RGB 十进制值			RGB 百分比			HSL		
		R	G	B	R	G	B	H	S	L
khaki	#f0e68c	240	230	140	94%	90%	55%	54	42%	75%
lavender	#e6e6fa	230	230	250	90%	90%	98%	240	8%	94%
lavenderBlush	#fff0f5	255	240	245	100%	94%	96%	340	6%	97%
lawnGreen	#7cfc00	124	252	0	49%	99%	0%	90	100%	49%
lemonChiffon	#fffacd	255	250	205	100%	98%	80%	54	20%	90%
lightBlue	#add8e6	173	216	230	68%	85%	90%	195	25%	79%
lightCoral	#f08080	240	128	128	94%	50%	50%	0	47%	72%
lightCyan	#e0ffff	224	255	255	88%	100%	100%	180	12%	94%
lightGoldenrod-Yellow	#fafad2	250	250	210	98%	98%	82%	60	16%	90%
lightGray	#d3d3d3	211	211	211	83%	83%	83%	0	0%	83%
lightGreen	#90ee90	144	238	144	56%	93%	56%	120	39%	75%
lightGrey	#d3d3d3	211	211	211	83%	83%	83%	0	0%	83%
lightPink	#ffb6c1	255	182	193	100%	71%	76%	351	29%	86%
lightSalmon	#ffa07a	255	160	122	100%	63%	48%	17	52%	74%
lightSeaGreen	#20b2aa	32	178	170	13%	70%	67%	177	82%	41%
lightSkyBlue	#87cefa	135	206	250	53%	81%	98%	203	46%	75%
lightSlateGray	#778899	119	136	153	47%	53%	60%	210	22%	53%
lightSlateGrey	#778899	119	136	153	47%	53%	60%	210	22%	53%
lightSteelBlue	#b0c4de	176	196	222	69%	77%	87%	214	21%	78%
lightYellow	#ffffe0	255	255	224	100%	100%	88%	60	12%	94%
lime	#00ff00	0	255	0	0%	100%	0%	120	100%	50%
limeGreen	#32cd32	50	205	50	20%	80%	20%	120	76%	50%
linen	#faf0e6	250	240	230	98%	94%	90%	30	8%	94%
magenta	#ff00ff	255	0	255	100%	0%	100%	300	100%	50%
maroon	#800000	128	0	0	50%	0%	0%	0	100%	25%
mediumAquama-rine	#66cdaa	102	205	170	40%	80%	67%	160	50%	60%
mediumBlue	#0000cd	0	0	205	0%	0%	80%	240	100%	40%
mediumOrchid	#ba55d3	186	85	211	73%	33%	83%	288	60%	58%
mediumPurple	#9370db	147	112	219	58%	44%	86%	260	49%	65%
mediumSeaGreen	#3cb371	60	179	113	24%	70%	44%	147	66%	47%

（续）

关键字	十六进制值	RGB 十进制值			RGB 百分比			HSL		
		R	G	B	R	G	B	H	S	L
mediumSlate-Blue	#7b68ee	123	104	238	48%	41%	93%	249	56%	67%
mediumSpring-Green	#00fa9a	0	250	154	0%	98%	60%	157	100%	49%
mediumTurquoise	#48d1cc	72	209	204	28%	82%	80%	178	66%	55%
mediumVioletRed	#c71585	199	21	133	78%	8%	52%	322	89%	43%
midnightBlue	#191970	25	25	112	10%	10%	44%	240	78%	27%
mintCream	#f5fffa	245	255	250	96%	100%	98%	150	4%	98%
mistyRose	#ffe4e1	255	228	225	100%	89%	88%	6	12%	94%
moccasin	#ffe4b5	255	228	181	100%	89%	71%	38	29%	85%
navajoWhite	#ffdead	255	222	173	100%	87%	68%	36	32%	84%
navy	#000080	0	0	128	0%	0%	50%	240	100%	25%
oldLace	#fdf5e6	253	245	230	99%	96%	90%	39	9%	95%
olive	#808000	128	128	0	50%	50%	0%	60	100%	25%
oliveDrab	#6b8e23	107	142	35	42%	56%	14%	80	75%	35%
orange	#ffa500	255	165	0	100%	65%	0%	39	100%	50%
orangeRed	#ff4500	255	69	0	100%	27%	0%	16	100%	50%
orchid	#da70d6	218	112	214	85%	44%	84%	302	49%	65%
paleGoldenrod	#eee8aa	238	232	170	93%	91%	67%	55	29%	80%
paleGreen	#98fb98	152	251	152	60%	98%	60%	120	39%	79%
paleTurquoise	#afeeee	175	238	238	69%	93%	93%	180	26%	81%
paleVioletRed	#db7093	219	112	147	86%	44%	58%	340	49%	65%
papayaWhip	#ffefd5	255	239	213	100%	94%	84%	37	16%	92%
peachPuff	#ffdab9	255	218	185	100%	85%	73%	28	27%	86%
peru	#cd853f	205	133	63	80%	52%	25%	30	69%	53%
pink	#ffc0cb	255	192	203	100%	75%	80%	350	25%	88%
plum	#dda0dd	221	160	221	87%	63%	87%	300	28%	75%
powderBlue	#b0e0e6	176	224	230	69%	88%	90%	187	23%	80%
purple	#800080	128	0	128	50%	0%	50%	300	100%	25%
rebeccaPurple	#663399	102	51	153	40%	20%	60%	270	67%	40%
red	#ff0000	255	0	0	100%	0%	0%	0	100%	50%
rosyBrown	#bc8f8f	188	143	143	74%	56%	56%	0	24%	65%

（续）

关键字	十六进制值	RGB 十进制值			RGB 百分比			HSL		
		R	G	B	R	G	B	H	S	L
royalBlue	#4169e1	65	105	225	25%	41%	88%	225	71%	57%
saddleBrown	#8b4513	139	69	19	55%	27%	7%	25	86%	31%
salmon	#fa8072	250	128	114	98%	50%	45%	6	54%	71%
sandyBrown	#f4a460	244	164	96	96%	64%	38%	28	61%	67%
seaGreen	#2e8b57	46	139	87	18%	55%	34%	146	67%	36%
seashell	#fff5ee	255	245	238	100%	96%	93%	25	7%	97%
sienna	#a0522d	160	82	45	63%	32%	18%	19	72%	40%
silver	#c0c0c0	192	192	192	75%	75%	75%	0	0%	75%
skyBlue	#87ceeb	135	206	235	53%	81%	92%	197	43%	73%
slateBlue	#6a5acd	106	90	205	42%	35%	80%	248	56%	58%
slateGray	#708090	112	128	144	44%	50%	56%	210	22%	50%
slateGrey	#708090	112	128	144	44%	50%	56%	210	22%	50%
snow	#fffafa	255	250	250	100%	98%	98%	0	2%	99%
springGreen	#00ff7f	0	255	127	0%	100%	50%	150	100%	50%
steelBlue	#4682b4	70	130	180	27%	51%	71%	207	61%	49%
tan	#d2b48c	210	180	140	82%	71%	55%	34	33%	69%
teal	#008080	0	128	128	0%	50%	50%	180	100%	25%
thistle	#d8bfd8	216	191	216	85%	75%	85%	300	12%	80%
tomato	#ff6347	255	99	71	100%	39%	28%	9	72%	64%
turquoise	#40e0d0	64	224	208	25%	88%	82%	174	71%	56%
violet	#ee82ee	238	130	238	93%	51%	93%	300	45%	72%
wheat	#f5deb3	245	222	179	96%	87%	70%	39	27%	83%
white	#ffffff	255	255	255	100%	100%	100%	0	0%	100%
whiteSmoke	#f5f5f5	245	245	245	96%	96%	96%	0	0%	96%
yellow	#ffff00	255	255	0	100%	100%	0%	60	100%	50%
yellowGreen	#9acd32	154	205	50	60%	80%	20%	80	76%	50%

附录B

元素，元素属性，样式属性

该指南给 SVG 渲染服务元素可用的属性及其默认值和可设置的值提供快速参考。

<linearGradient>

颜色结点绘制为与渐变矢量垂直（在渐变的坐标系内）的平行线的渐变。

`id`

它的值用于引用该渐变。

- 与所有其他元素 `id` 的限制相同。

`x1`

渐变矢量起点的水平位置。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系宽度）。
- 默认值为 0。

`y1`

渐变矢量起点的垂直位置。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系高度）。
- 默认值为 0。

x2

渐变矢量终点的水平位置。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系宽度）。
- 默认值为 100%。

y2

渐变矢量终点的垂直位置。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系高度）。
- 默认值为 0。

gradientUnits

使用的坐标系。

- 值为 userSpaceOnUse 或 objectBoundingBox。
- 默认值为 objectBoundingBox。

gradientTransform

应用在渐变内容的变换，独立于它要填充的形状。

- 空白分隔的变换函数列表：translate(tx,ty)、scale(s)、scale(sx, sy)、rotate(a)、rotate(a,cx,cy)、skewX(a) 和 skewY(a)。
- 每个变换的参数都是不带单位的数字（在 SVG 1 的语法中）；长度被解析为用户单位（px），角度被解析为度数。
- 默认值为没有变换。

spreadMethod

用在超出渐变矢量起点和终点的填充内容的策略。

- 值为 pad、reflect 或 repeat。
- 默认值为 pad。

xlink:href

引用另一个渐变来作为当前渐变的模板。

- 指向目标片段的 URL，它的值必须匹配相同文档内 <linearGradient> 或 <radialGradient> 元素的 ID。
- 引用的元素的所有属性作为当前元素的默认值。
- 如果当前元素不包含任何 <stop> 元素，使用引用的元素的结点。

- 在 XML 文档中（包括 SVG），`xlink` 前缀必须附属于 XLink 命名空间，*http://www.w3.org/ 1999/xlink*，使用 `xmlns:xlink` 属性。

<radialGradient>

颜色结点沿着起点向外部圆发散的渐变。

`id`

　它的值用于引用该渐变。

- 与所有其他元素 `id` 的限制相同。

`cx`

　中心点的水平位置。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系宽度）。
- 默认值为 50%。

`cy`

　中心点的垂直位置。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系高度）。
- 默认值为 50%。

`r`

　圆的半径。

- 一个长度值（在用户坐标系内或有单位）或百分比（与坐标系对角线长度成比例，使对角线始终为 $\sqrt{2}$ ×100%）。
- 默认值为 50% 。
- 负值会报错。

`fx`

　焦点的水平位置。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系宽度）。
- 默认值与 `cx` 的值相同。

fy

焦点的垂直位置。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系高度）。
- 默认值与 cy 的值相同。

gradientUnits

使用的坐标系。

- 值为 userSpaceOnUse 或 objectBoundingBox。
- 默认值为 objectBoundingBox。

gradientTransform

应用在渐变内容的变换，独立于它要填充的形状。

- 语法和可选值与 <linearGradient> 相同。

spreadMethod

用在超出最外层圆的填充内容的策略。

- 语法和可选值与 <linearGradient> 相同。

xlink:href

引用另一个渐变来作为当前渐变的模板。

- 语法和可选值与 <linearGradient> 相同。

<stop>

渐变内的一个固定值。

offset

沿着渐变矢量或射线定位该值对应的距离。

- 0 到 1 之间的数字或百分数。
- 它的值被限制在 [0~1] 或 [0%~100%] 的范围内。
- 结点必须按照偏移增长的顺序排列，如果不这样，偏移值会被调整为之前的最大值。
- SVG 1.1 中官方没有说明默认值；SVG 2 和大多数浏览器中默认值为 0。

`stop-color`（**表现属性**）

用在该结点的颜色。

- 任何浏览器支持的合法的颜色定义。
- 默认为 `black`。

`stop-opacity`（**表现属性**）

用在该结点的 `alpha` 值。

- 0 到 1 之间的数字。
- 默认值为 1。

<pattern>

一个渲染服务，它定义一个用于填充其他元素的自定义 SVG 内容的区域，根据需要在矩形网格中进行重复。

`id`

它的值用于引用该渐变。

- 与所有其他元素 `id` 的限制相同。

`x`

相对于图案瓷砖左上角的水平偏移。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系宽度）。
- 默认值为 0。

`y`

相对于图案瓷砖左上角的垂直偏移。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系高度）。
- 默认值为 0。

`width`

每个图案瓷砖的宽度。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系宽度）。
- 默认值为 0，此时禁止渲染图案。

- 负值会报错。

`height`

每个图案瓷砖的高度。

- 一个长度值（在用户坐标系内或有单位）或百分比（相对于坐标系高度）。
- 默认值为 0，此时禁止渲染图案。
- 负值会报错。

`patternUnits`

x、y、width 和 height 使用的坐标系。

- 值为 userSpaceOnUse 或 objectBoundingBox。
- 默认值为 objectBoundingBox。

`patternContentUnits`

绘制图案子内容所使用的坐标系。

- 值为 userSpaceOnUse 或 objectBoundingBox。
- 默认值为 userSpaceOnUse。
- 如果指定了 viewBox 则没有效果。
- 值为 objectBoundingBox 时实现为对用户单位非均匀缩放，它不会为百分比长度创建一个新的参照。

`viewBox`

声明一个用于图案内容的自定义坐标系。

- 四个数字的列表，以空格或逗号分隔。
- 数字代表的值，依次为：最小的 x，最小的 y，宽，高。
- 宽度值和高度值必须为正。
- 默认情况下，坐标系通过 patternContentUnits 来控制。
- viewBox 实现为一个缩放变换,且不会为百分比长度创建一个新的参照。

`preserveAspectRatio`

当 viewBox 定义的宽高比与图案瓷砖的宽高比不匹配时使用的缩放和对齐策略。

- 值为 none 或者一个对齐值后紧跟 meet 或 slice。
- 对齐值是一个单独的单词 xM*xx*YM*xx*，其中 M*xx* 为 Min，Mid，Max 之一。
- 默认值为 xMidYMid meet。
- 只有在指定了 viewBox 属性时才有效果。

`patternTransform`

应用在图案瓷砖和图案内容的变换，独立于它要填充的形状。

- 语法和可选值与 `<linearGradient>` 的 `gradientTransform` 属性相同。
- 默认值为没有变换。

`xlink:href`

引用另一个图案来作为当前图案的模板。

- 指向目标片段的 URL，它的值必须匹配相同文档内 `<pattern>` 元素的 ID。
- 引用的元素的所有属性作为当前元素的默认值。
- 如果当前元素不包含任何子元素，则使用引用的元素的图案内容。
- 在 XML 文档中（包括 SVG），`xlink` 前缀必须附属于 XLink 命名空间，*http://www.w3.org/ 1999/xlink*，使用 `xmlns:xlink` 属性。

作者介绍

Amelia Bellamy-Royds 是一位科技领域内的自由撰稿人。她在 Web 设计圈内因关于 SVG 的著作而出名。Amelia 是 W3C SVG 工作组的特邀专家，并且积极参与 SVG 辅助工作组的工作。她通过参与在线社区（例如 Web Platform Docs、Stack Exchange 和 Codepen）来帮助促进 Web 标准和设计的发展。

Amelia 对 SVG 的兴趣来自于数据可视化的工作，并建立在她获得生物信息学理学学士时学到的编程基础之上。从那以后，她开始从事科学、健康和环境政策的研究，且获得了新闻学硕士学位。Amelia 目前居住在加拿大的艾伯塔省埃德蒙顿市。如果她不在电脑旁，很可能在打理她的菜园或外出享受现场音乐。

Kurt Cagle 曾经是 SVG 工作组的成员，且在 2004 年出版了第一个本关于 SVG 的书。目前是作为 W3C Xforms 工作组的特邀专家，他在为美国国家档案馆工作之后，成为国会图书馆的 XML 数据架构师。他从 2003 年以来一直是 O'Reilly Media 的定期投稿者，并且还在 2008 年至 2009 年期间担任在线编辑。

封面介绍

本书封面上的动物是藏族血雉（*Ithaginis cruentus tibetanus*）。这种小的，像鹧鸪一样的雉鸡在整个不丹东部和西藏南部山区的森林地区数量稳定。整个夏天可以在高海拔地区发现它们，秋天和冬天随着降雪量的增加它们会退回到山谷。

血雉上身羽毛是蓝灰色的，下身有一些苹果绿的羽毛。该物种的通用名称指的是它们胸部的羽毛，呈典型的明亮的深红色，就像血迹一般。这些动物还可以通过红色的脚和红色的眼睛来区分。

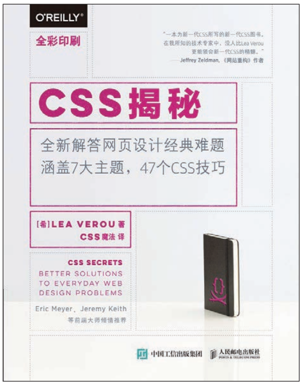
雄性通常 1.5 英尺长，雌性略小，但颜色更加柔和、均匀。它们的强壮的爪子用于抓取食物，主要是绿色植物，如苔藓、蕨类植物和松树枝。

血雉并不特别擅长于飞行，它们会在 4 月下旬和 5 月初开始在地面上筑巢。雄性守护雌性孵化蛋（通常六或七个）。如果它们感觉到了危险，就会将巢穴移到一个新的位置或遗弃。小鸡会在 6 月中旬孵出，并且和它的妈妈在一起直到冬天。

O'Reilly 封面上的许多动物都濒临灭绝，它们对于世界来说非常重要。要想了解更多关于如何提供帮助的信息，请访问 *animals.oreilly.com*。

封面上的动物是 Karen Montgomery 基于 *Wood's Natural History* 中的一个雕刻品所做。

技术改变世界 · 阅读塑造人生

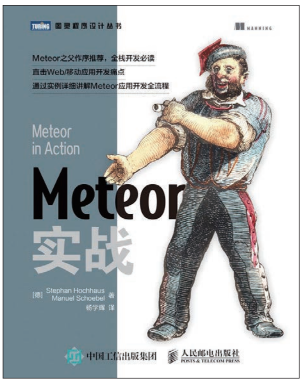


CSS 揭秘

- CSS一姐Lea Verou作品
- 近年来最重要的CSS技术书
- 全新解答网页设计经典难题

作者： Lea Verou

译者： CSS 魔法

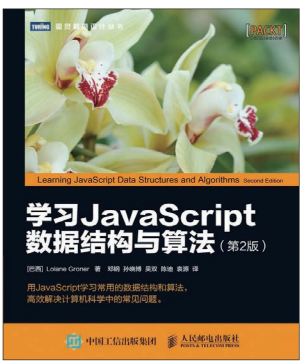


Meteor 实战

- Meteor之父作序推荐，全栈开发必读
- 直击Web/移动应用开发痛点
- 通过实例详细讲解Meteor应用开发全流程

作者： Stephan Hochhaus , Manuel Schoebel

译者： 杨学辉



学习 JavaScript 数据结构与算法（第 2 版）

- 用JavaScript学习常用的数据结构和算法，高效解决计算机科学中的常见问题

作者： Loiane Groner

译者： 邓钢等

O'REILLY®

深入理解SVG

作为W3C的开放标准，SVG被越来越多的开发者所关注。SVG不只是简单的矢量图，还可以加上更复杂的绘画和更细致的效果，包括渲染、渐变、应用到文本，甚至可以添加照片。

本书深入介绍SVG绘画。主要内容如下。

- **SVG渲染模型如何实现描边和填充**
- **标准颜色的应用，自定义颜色，创建颜色模板**
- **透明度的设置**
- **如何控制线性渐变在要渲染的形状内移动**
- **重复线性渐变**
- **磁贴、纹理和图片图案**
- **如何在文本上应用渲染服务**
- **给渲染服务添加动画**

"本书详尽介绍SVG渲染和模型特性，给你不一样的Web绘图体验！"

——Erik Dahlström
SVG工作组联合主席

Amelia Bellamy-Royds
W3C SVG工作组特邀专家，积极参与SVG辅助工作组的工作，并通过Web Platform Docs等在线社区来帮助促进Web标准和设计的发展。

Kurt Cagle
SVG工作组成员，目前是W3C Xforms工作组特邀专家，美国国会图书馆XML数据架构师。

XML/WEB DESIGN

封面设计：Ellie Volckhausen 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／程序设计／前端开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-46758-4
9 787115 467584 >
ISBN 978-7-115-46758-4
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks